Reactive Programming in Ada 2012 with RxAda*

Alejandro R. Mosteo

Centro Universitario de la Defensa de Zaragoza, Carretera de Huesca s/n, 50090, Zaragoza, Spain
Instituto de Investigación en Ingeniería de Aragón, C/ Mariano Esquillor s/n, 50018, Zaragoza, Spain

ARTICLE INFO

Keywords:
ReactiveX
Observer Pattern
Reactive Programming
Ada 2012

Abstract

The ReactiveX API, also known as the Reactive Extensions in the .NET world, is a popular functional reactive programming framework for asynchronous, event-based, multithreaded programming. Although Ada built-in tasking reduces the dire needs for additional multithreading support of some other languages, the reactive approach has properties that are well-suited to the safety and maintainability culture predominant in the Ada world, such as complexity reduction, well-defined concurrency semantics, and enhanced legibility by means of concise and explicit information flows appealing to imperative reasoning. This work presents the design of a ReactiveX Ada implementation that aims to balance desirable library properties such as compile-time type-safety, amount of user-required generic instantiations, and a smooth learning curve for both library clients and maintainers. Concurrency design aspects of the library are detailed, showing how the `Flat_Map` and `Thread` abstractions have been implemented following Ada programming expectations, in particular with regard to task termination. In the intervening time from its first presentation, the library has gained implemented operators to the point of having all fundamental building blocks available. With RxAda, the Ada programmer can henceforth benefit from the abundant documentation existing for the language-agnostic ReactiveX approach without stepping out of the Ada tool chain.

1. Introduction

Modern applications are becoming increasingly complex, in many cases driven by external events with unpredictable latencies caused by user interaction, external sources of information, or remote components in distributed systems, for example. Such changes, in turn, require modifications to local states and may generate new internal or remote events. The reactive programming paradigm [12] arises as a response to the challenge of implementing such systems, in which imperative languages have shown shortcomings: the traditional model in which the program imposes the control flow is reversed, becoming a loop that waits for events to which it is necessary to react. This inversion of control presents challenges [1], like poor understanding by novice programmers, unresponsive systems that fail to exploit concurrency, convoluted state management, among others. A traditional approach has been callbacks, which hamper scalability. The interactions between callbacks through shared states can rapidly become too complex, and callbacks themselves can be spread through many locations, complicating maintenance. The term *callback hell* [4] is not unheard of.

The reactive paradigm is not particularly novel; formalization efforts have been shown [17], and proposals using the Ada syntax have been described [16]. It is, however, relatively recently that reactive programming has become popular, with examples like the .NET standard reactive extensions [14], subsequently ported to many languages, and the publication of the Reactive Manifesto [2]. At least part of the appeal in reactive programming is the ability for the imperative programmer to represent logical sequences much like in typical imperative syntax, while retaining control of the concurrency involved, and without requiring complex state management.

By generalizing the observer pattern [6], in contraposition to the iterator pattern [8], the ReactiveX approach [14] to reactive programming provides composable abstractions [5, 9] that allow programmers to represent responses to events as complete information flows. These flows or sequences transform data in apparent imperative fashion, thanks to their declarative style. The configurable asynchronicity of flows in regard to their point of declaration allows the programmer to dispense with blocking concerns from particular threads (typically the user interface thread). Mutual exclusion guarantees within a flow, in turn, can be relied on when designing the multithreading architecture of applications.

This work presents a high-level port of the ReactiveX framework to the Ada 2012 language, named as RxAda [10, 11], focusing on design aspects of the implementation. Compared to the work in [11], this article delves deeper in the concurrency design of the library, detailing aspects of its `Flat_Map` and `Thread` implementation, and also presents several concurrency patterns through a practical application consisting on hashing the files found in a folder tree, using standard Rx operators and schedulers already implemented in RxAda.

The current version of Ada lacks functional facilities of other languages like lambda functions and implicit instantiation. These limitations challenge a practical ReactiveX implementation which have been addressed by means of a combination of object-oriented and generics-based design. The article is written assuming some knowledge of Ada 2005 and no prior knowledge of the ReactiveX framework (although familiar readers will be able to contrast the RxAda idioms versus other language implementations). The focus is

* This work was supported by the Spanish projects DPI2016-76676-R-AEI/FEDER-UE, CUD2018-03, CUD2019-05, DGA-T45_17R/FSE, DGA-T45_20R/FSE.

ORCID(s): 0000-0001-7853-3622 (A.R. Mosteo)

Listing 2.1: Ada interfaces for the Rx contract.

```
generic
    type T (<>) is private; -- T is the user type to be pushed down
package Rx.Contracts is

    type Observer is interface;
    -- Someone interested in receiving data.

    procedure On_Next     (This  : in out Observer; V : T)
    is abstract;
    -- Delivers one item.

    procedure On_Complete (This  : in out Observer)
    is abstract;
    -- Called upon subscription completion.

    procedure On_Error    (This  : in out Observer;
                           Error :        Errors.Occurrence)
    is abstract;
    -- Errors encapsulate an upstream Exception_Occurrence.

    type Observable is interface;
    -- An emitter of data to which an observer can subscribe.

    procedure Subscribe (Producer : in out Observable;
                         Consumer : in out Observer'Class)
    is abstract;
    -- Begins a subscription on the upstream source observable.
end Rx.Contracts;
```

placed on aspects of the library that are relevant to prospective users, and that could also be interesting to Ada architects and practitioners in general. The examples for common concurrency patterns highlight how the RxAda way can supplement traditional Ada multitasking features.

The article is structured as follows: Section 2 introduces with examples the basics of the ReactiveX framework. Section 3 discusses RxAda design challenges and the solutions adopted with its advantages and drawbacks. Concurrency design details and use is seen through examples using DirX, a RxAda-based alternative to `Ada.Directories`, in Section 4. Next, Section 5 presents library organization details of relevance to users and maintainers. Lastly, concluding remarks close the article in Section 6.

2. Reactive Extensions Overview

The definitions of the main concepts that transpire the ReactiveX API are presented now, before some introductory examples[1]. In the following presentation, italized words are Rx-specific jargon with precise meaning, whereas fixed-size font is used for Rx types and subprograms. Since RxAda has followed where possible the RxJava specification [7], its documentation would be the most useful to new RxAda users. Also, some Java examples are provided for comparison.

The foundation of the ReactiveX approach are the `Observable` and `Observer` interfaces, along with the Rx grammar (also referred to as the reactive contract [15]). An observer *subscribes* to an observable, after which it may receive at any time a new datum (an *item*) from the observable via a call to the observer `On_Next` subprogram. The RxAda implementation of these interfaces is shown in Listing 2.1.

[1] In-depth documentation is available at the official website [14].

Listing 2.2: Separate chain building and subscription.

```
declare
    S1, S2 : Rx.Subscriptions.Subscription;
          -- A subscription is returned when subscribing to allow
          -- asynchronous premature cancellation of a flow.
    Chain  : Rx.Std.Integers.Observable'Class :=
        Rx.Std.Integers.From (1, 2, 3, 4, 5)
          -- "From" emits these five integers in sequence.
        & Rx.Std.Integers.Filter (Is_Even'Access)
          -- "Filter" drops items not passing the test
          -- function Is_Even (I : Integer) return Boolean
          -- declared elsewhere.
        & Rx.Std.Count;
          -- "Count" emits the count of items received when
          -- On_Complete arrives, and then completes itself.
begin
    S1 := Chain
         & Subscribe (On_Next => Rx.Debug.Print'Access);
         -- Will print a 2.
    S2 := Chain
         & Subscribe (On_Next => Rx.Debug.Print'Access);
         -- Will print a 2, too.
end;
```

Per the reactive contract (in POSIX-like regular expression syntax),

$$\texttt{On_Next}^{*}\,(\,\texttt{On_Complete}\,|\,\texttt{On_Error}\,)?,$$

after subscribing, the observer may receive any number of `On_Next` calls (including none), possibly followed by either `On_Complete`, to mark the end of the sequence, or `On_Error`, if something untoward happened upstream, but not both, and never more than once. Also part of the contract is that these three methods will always be called in mutual exclusion in a given observer, thus freeing users from concerns with concurrent access to local state. Since data propagation is performed by the observable calling `On_Next` on its observer, Rx is a push-based framework. Observers cannot know when a new item will arrive, nor can they request items at will.

Although superficially similar to traditional callback programming with an enriched dynamic behavior, the true expressiveness of the Rx approach emanates from its *operators*, which are themselves both observables and observers that can be composed one after another, with each operator implementing a modification to be applied to the items traveling through them. In other words, a chain or sequence of observables can be built, rooted at some source observable. When an observer subscribes to this chain, the root observable will begin emitting data by calling the `On_Next` in the next operator in the chain. Operators apply their action and push down the item until it reaches the subscriber.

A first example of a chain is shown in Listing 2.2, and comparable examples in both Java and Ada are presented side-by-side in Listing 2.3. It is worth stressing that concatenating operators does not trigger a subscription. Thus, operators are passive elements that by themselves do not cause an observable to start emitting items[2]. Also, the operators code is not executed until a subscription is performed.

Listing 2.2 exemplifies the two separate phases in the process of chain building and subscription. Each subscrip-

[2] A related Rx concept that does create a subscription is a `Subject`, that is out of the scope of this introduction. This is of importance for *cold/hot* observables, which is another Rx concept left out of this introduction.

Listing 2.3: Rx Java/Ada example. Appropriate functions in the Ada example are assumed to exist, and "use" clauses are omitted. The complete example can be found online at `https://github.com/mosteo/rxada/blob/jsa2019/src/main/rx-jsa2019.adb`.

```java
{
 rx.Observable
    .interval(1, TimeUnit.SECONDS)
        // The interval observable is a counter that emits successive
        // values separated by the given time period in a
        // background Thread.
    .observeOn(Schedulers.computation())
        // Switch the data flow to a computation thread.
    .map(Object::toString)
        // Method reference notation.
    .map(s -> s.hashCode())
        // Lambda notation.
    .observeOn(Schedulers.io())
        // Switch to an Input/Output thread.
    .subscribe(System.out::println);
}
// Java allows ignoring the returned subscription, whereas the Ada
//   example has to explicitly capture it.
// Java benefits from implicit instantiation and in-line lambda
//   expressions, introduced with Java 8.
```

```ada
declare
  S : Subscription :=
    Interval (First => 1, Period => 1.0)
        -- The RxAda Interval observable uses Duration as the time
        -- unit, and uses Ada tasks to implement Rx threads.
    & Observe_On (Schedulers.Computation)
        -- Switch to a computation task.
    & Map (Image'Access)
        -- Image takes an Integer and returns its String image.
    & Map (String_Hash'Access)
        -- E.g. instance of System.String_Hash,
        -- returns an Integer.
    & Observe_On (Schedulers.Input_Output)
        -- Switch to an input/output thread.
    & Subscribe (Put_Line'Access);
begin
    null; -- At this point the previous chain is already subscribed
          -- and hence active.
end; --  In RxAda, lambda functions are replaced by either accesses
     --  to functions or overridable interfaces from Rx.Actions.
```

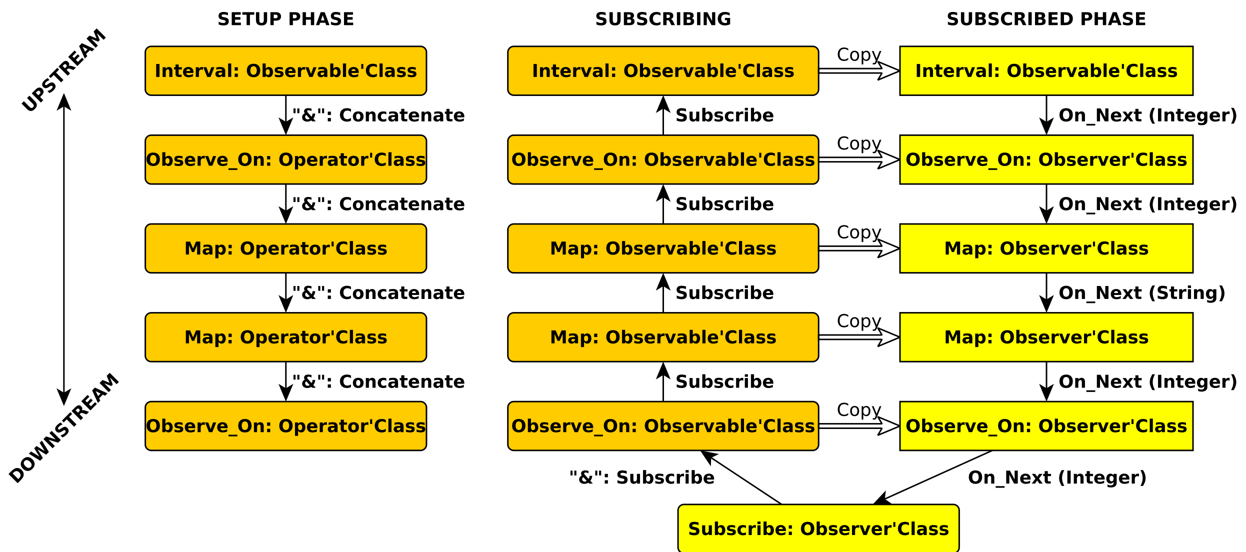

Figure 1: Anatomy of a data chain in RxAda (based on Listing 2.3). The relevant interfaces for each stage are indicated after the object name. Rounded boxes are used to indicate passive chains, whereas sharp ones indicate a live subscription.

tion causes the `From` observable to emit its items, and both subscribers will see the same final count, since a fresh `Count` operator instance is created for each subscription. A consequence is that chain building is synchronous (that is, it happens as the program execution reaches that point), but the subscription flow may be asynchronous to the program flow, which is sometimes a confusing point for beginners. For example, in Listing 2.3, one item is emitted per second, in some unspecified Rx task, whereas the main program task can be anywhere else. Another important property is that fresh operators are used for each subscription, meaning that both subscribers in Listing 2.2 will see the same final count instead of a cumulative count. This initialization property allows the preparation of partial pipelines that can be later interconnected, attached to observables, and finally subscribed when convenient, with pristine operator chains for every subscription.

Figure 1 details these aspects of Rx operation. During the setup phase (leftmost column), operators are concatenated one after the other with the "&" function, hence the program flow goes from top to bottom. When a consumer observer (sink) subscribes (center) using the overloaded "&" function, the subscription progresses from bottom to top via `Subscribe` calls until reaching the source observable, that emits the items. It is at this time that operators are copied and initialized, so that each subscribed chain is made of pristine operators. Once the subscribed phase starts, item propagation takes place from source observable to subscribed observer, with each operator in turn applying its operation before pushing down the item.

Another characteristic is that Rx is lazy in regard to task creation [15], so the user should assume that tasks are reused unless explicitly requested by scheduling operators or otherwise necessary for the operator proper working (as in the

Listing 3.1: Map operator definition in RxJava

```java
public class Observable<T> {
    // T is the type emitted by this observable.

    public final
        <R> Observable<R> map(Func1<? super T,? extends R> func);
    // R is the result of "func", and the type emitted by the
    // observable returned by "map". Hence, each T produces an R.

    // Other operators omitted.
}
```

Listing 3.2: Possible definition not requiring generic user types

```ada
package Rx.Contracts is -- No longer generic.

    type Rx_Item is interface;

    type Observer is interface;
    --  Someone interested in receiving data.

    procedure On_Next (This : in out Observer; V : Rx_Item'Class)
    is abstract;
```

`Interval` example in Listing 2.3). That is, in the example in Listing 2.2, all data emission, filtering, counting and output will happen in the task that performs the subscription, because no scheduling operators like `Observe_On` have been used.

Going back to the basic example in Listing 2.3, the main visual difference is that Java uses dot notation to create Rx chains (also called flows), whereas in Ada the "&" operator has been chosen. The reasons will become apparent in the next section, stemming from the Ada freezing rules for primitive tagged type operations.

To summarize this Rx introduction:

- Any data source that can be wrapped as an observable can serve as the root of a chain.
- There are two distinct *setup* and *subscribed* phases.
- A chain is a pipeline through which items are sent, in mutual exclusion, to subscribers.
- Operators are composable and, during the subscribed phase, perform operations on the items, either to each one individually or doing some kind of reduction.
- Each subscribed chain is built with new operator instances, seeing the same initial state from Rx operators as other subscriptions made to the same chain.
- Once a flow completes or errs, no further data will reach the downstream subscriber.

The takeaway from this Rx introduction is that operator chains allow the representation of item transformations in a naturally ordered flow with cohesive temporal logic that includes task switching. Hence, operators can take as much time as needed without blocking concerns and spaghetti callback jumps, letting the user focus on the application logic.

3. RxAda Design

This section discusses some all-pervasive design decisions. More precisely, the generic model of the library is first introduced. This model in turn affects the library implementation facilities and the way clients can use the library. Concurrency aspects of the notable `Merge` and `Flat_Map` operators are presented next. The general dependency architecture of RxAda packages is summarized last.

3.1. Typed Operators

As evidenced by Listing 2.1, observers receive items of a single type in `On_Next` calls. Observables, in turn, must be aware of this type to be able to call `On_Next` on subscribers. Seemingly nothing untowards (any collection library has the stored type as a generic formal), the challenge arises from operators that transform the type being emitted. Operators exhibit both observable and observer interfaces, but not necessarily of the same type. For example, the `Map` operator allows the conversion or processing of a type into another by means of a function parameter that, given the upstream incoming type, returns a possibly different type that is passed downstream. Listing 3.1 shows the Java specification.

While such a specification is par for the course with implicit instances in Java or C++, in Ada this requires explicit instances with two formal parameter types, or two instances with the second one nested inside the first one. While explicit instances may be clearer about intended code purpose, they are in this case an obstacle to the use of dot notation, which in Ada only works with primitive subprograms. This issue is further explored in Sec. 3.2.

An alternative design was considered: a root interface type could be declared, like demonstrated in Listing 3.2. This approach is superficially tempting since it removes the need for user instantiations when using the library, allowing the use of dot notation for all operators. The price to pay is the lack of type consistency checks between chained operators at compile time, since all of them would deal with the same `Rx_Item'Class` classwide data.

Furthermore, user types would have to be made descendants of the root `Rx_Item` interface, which is a distributed pollution imposed outside of the library that may discomfort users. In addition, user functions should either perform casts that could fail at runtime or generic marshallers should be instantiated for convenience (that could not be compile-time checked anyway).

This approach was not adopted for the stated shortcomings, and consequently one instantiation is needed per user type involved in a chain. Since some operators emit known data types (e.g., `Count` emits integers), `String`, `Integer` and `Float` types come preinstantiated and ready for use in the package `Rx.Std`.

The materialization of a user type, from an RxAda internal point of view, is in package `Rx.Impl.Typed`. This package is used as a generic formal through the rest of the implementation, and contains information about a user type in

Listing 3.3: RxAda root package for a user type

```
generic
    with package Type_Traits is new Rx.Traits.Types (<>);
    -- Traits are used to efficiently work with both
    -- definite and indefinite user types.
package Rx.Impl.Typed is

    subtype T is Type_Traits.T; -- T (<>) is the user supplied type.

    -- Following packages are typed instances for the user type T.

    package Contracts is new Rx.Contracts (T);
    package Actions   is new Rx.Actions.Typed (T);

    subtype Observable is Contracts.Observable;
    subtype Observer   is Contracts.Observer;

    --  And more...
```

Listing 3.4: RxAda root package for operators

```
generic
   with package From is new Rx.Impl.Typed (<>); -- Upstream
   with package Into is new Rx.Impl.Typed (<>); -- Downstream
package Rx.Impl.Transformers is

   -- The package receives its name from the fact that the
   -- input and output types can be different.

   type Operator is new           -- Base null operator helper type.
      From.Contracts.Observer and -- Observes data of From.T type.
      Into.Contracts.Observable   -- Emits data of Into.T type.
      with private;

   overriding
   procedure On_Next (This : in out Operator;
                      V    :        From.T)
                      is abstract;
   -- Observes pushed data of From.T type.

   overriding
   procedure Subscribe (This     : in out Operator;
                        Observer : in out Into.Observer'Class);
   -- Can be subscribed to by observers of Into.T type.

   function "&" (Producer : From.Contracts.Observable'Class;
                 Consumer : Operator'Class)
                 return Into.Contracts.Observable'Class;
   -- With asymmetric "&", type-checked chains can be set up.
   -- This operator is also renamed as Concatenate.
```

relation with the Rx contract that other parts of the RxAda implementation require. A traits-based approach [3] is used for user types, enabling storage control. See Listing 3.3 for details.

3.2. Type Transformations

Once established that observers are statically typed, the issue of chaining operators arises. In Java, as seen in the `Map` example, operators are a primitive operation of the Observable class. In Ada, primitive operations must be declared in the same package as the type. Hence, to use the same dot notation for operator chaining, all operators should be declared within the same Observable package. Unfortunately, when two types are involved, a second instantiation is required, which no longer provides primitive subprograms. Two-type (observable/observer) generic packages would not work either, because the downstream type must be able to become the upstream type, and generics in Ada cannot reference instances of themselves, or circularly refer to themselves through a chain of generic specifications.

Listing 3.5: Detail of "&"-related types with explicit package names

```
-- A fictitious syntax is used in the following comments to indicate
-- both the base interface of the types and the user type (between
-- parentheses) with which the packages are instantiated (through
-- instances of package Typed).

-- package Rx.Std contains instances for Integer and String types.

package Integers renames Rx.Std.Integers;
package Strings  renames Rx.Std.Strings;
package Int_To_Str renames Rx.Std.Integer_To_String;
package Str_To_Int renames Rx.Std.String_To_Integer;

S : Rx.Subscriptions.Subscription :=
    -- Allows termination and liveness checking.
    Integers.Interval (First => 1, Period => 1.0)
        -- Contracts (Integer).Observable'Class
    &   -- Concatenation using Preservers (Integer)."&"
    Integers.Observe_On (Rx.Schedulers.Computation)
        -- Preservers (Integer).Operator'Class
    &   -- Concatenation using Transformers (Integer, String)."&"
    Int_To_Str.Map (Image'Access)
        -- Transformers (Integer, String).Operator'Class
    &   -- Concatenation using Preservers (String)."&"
    Str_To_Int.Map (String_Hash'Access)
        -- Transformers (String, Integer).Operator'Class
    &   -- Concatenation using Preservers (Integer)."&"
    Integers.Observe_On (Schedulers.IO)
        -- Preservers (Integer).Operator'Class
    &   -- Subscription using Contracts (Integer)."&"
    Integers.Subscribe (Put_Line'Access);
        -- Contracts (Integer).Sink'Class
```

These reasons preclude a natural Ada solution that uses dot notation. The adopted solution in RxAda is the use, as in the C++ implementation, of a binary operator function. Whereas C++ uses the pipe "`|`" operator, in Ada the "&" operator was chosen, which furthermore already conveys the sense of concatenation to Ada programmers. This is realized as seen in Listing 3.4 where the root Operator class of RxAda is defined.

As evidenced by the parameters accepted by "&", users have to perform an instantiation for every conversion between types, in the proper `From`→`Into` direction of transformation. And, since "&" is defined for two precise types, the proper consistency checks for operators forming a chain are performed at compile time. The second parameter of "&" is returned under its Observable'Class view, to be used as the first parameter of a subsequent "&" call. Listing 3.5 shows again the initial example of Listing 2.3, this time detailing the different types involved.

3.3. Type-Preserving Operators

While, in the general case, operators such as `Map` involve two different types for upstream and downstream, many other operators are only meaningful within a single type. For example, `Filter` lets items through depending on some test performed on the item. In the particular case of these operators, both upstream and downstream types match. In RxAda the `Preserver` class is the `Transformer` specialization for such case, as shown in Listing 3.6.

Listing 3.6: Specialization for type-preserving operators

```
generic
   with package Typed is new Rx.Impl.Typed (<>);
   -- Where in Transformer From and Into had to be provided, here
   -- Typed suffices, since both received and emitted types are
   -- one and the same.
package Rx.Impl.Preservers is

   package Transform is new Rx.Transformers (Typed, Typed);
   -- The same type is received and emitted.

   subtype Operator is Transform.Operator; -- Just a shortcut.

   function "&" (Producer : Typed.Contracts.Observable'Class;
                 Consumer : Operator'Class)
                 return Typed.Contracts.Observable'Class
   renames Transform. "&";
```

Listing 3.7: Subscription-related declarations

```
type Sink is abstract new Observer with private;
-- Specialized type to recognize subscription intent and store a
-- shared Subscription. Such subscription is returned by the
-- following "&" function.

function "&" (Producer : Observable'Class;
              Consumer : Sink'Class) return Subscription;
-- The actual function that calls to Producer.Subscribe (Consumer).

function Subscribe (Using : Observer'Class) return Sink'Class;
-- Function that wraps a regular Observer into a Sink.
-- A version taking procedure accesses for On_Next, On_Complete and
-- On_Error is also available. See Rx.Subscribe for details.
```

3.4. Executing the Subscription

Previous examples ended the chain with the `Subscribe` function (e.g., Listing 3.5). In RxAda, to distinguish between a regular operator concatenation and an actual subscription, and for uniformity, the “&” symbol is used with a different parameter profile. As shown in Listing 3.7, a specific `Sink` interface is used that disambiguates for the compiler the precise “&” being called. The returned `Subscription` is in practice a hidden reference-counted pointer to a shared atomic boolean that can be used to externally and asynchronously terminate a subscription, or check its liveness.

3.5. Dependencies and User Instantiations

To conclude this section, dependencies between the already seen parts of the library are graphically depicted in Figure 2. Going from bottom to top, `Rx.Contracts` declares the interfaces for the Reactive Contract, whereas the generic package `Rx.Traits.Types` enables the user to specify a definite representation for an indefinite type, and the proper conversions. The `Rx.Impl.Typed` package takes a user type and combines it with the Rx contract via implementation packages, not shown in the figure. The rest of `Rx.Impl.*` packages depicted are successive specializations of the Operator implementation infrastructure. The `Rx.Op.*` packages are the actual Rx operator implementations, which in turn can be instantiated directly by an advanced user, or through the convenience packages seen in the top layer.

At this point the reader might rightfully wonder how many

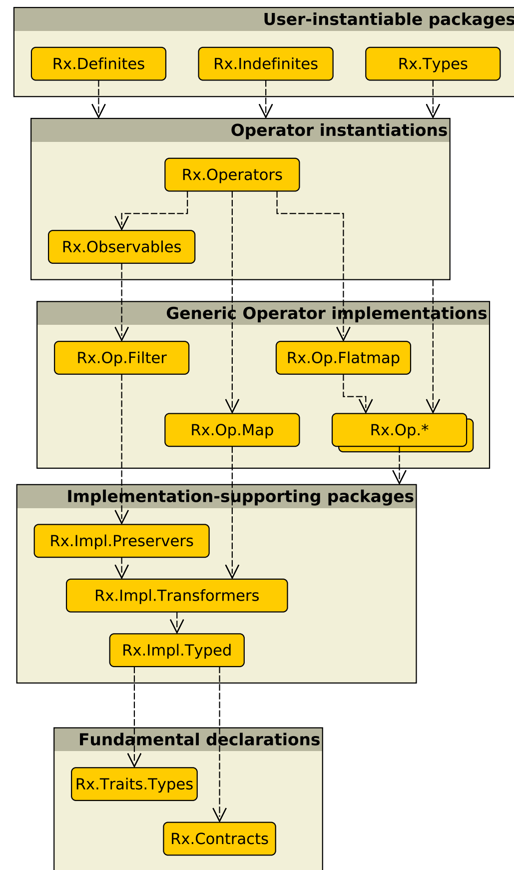

Figure 2: Dependencies between some RxAda packages. This partial view of the package hierarchy represents the basic interactions described in Section 3.

and of which package instances should be created to be able to use RxAda. To ease the initial learning curve and for simple use cases, RxAda provides two packages that take care of the finer details with sensible defaults, so the new user needs only to choose between `Rx.Definites` or `Rx.Indefinites` as the entry point into RxAda. These packages take as formal only the user type (see Listing 3.8) and create instances of all single-type observables, ready for use. Finer storage control is available through `Rx.Types`. The `Rx.Operators` package, in turn, instantiates operators that transform between types (called also transformers in RxAda). For even finer control, the user can dive into the individual operator packages, whose organization is described in Section 5.

4. Concurrency in RxAda

This section delves into details of concurrent programming with RxAda, relying on an example using DirX. This companion library enables the use of `Ada.Directories` with RxAda, by providing observables for the standard-defined `Directory_Entry_Type` values. Before that, some necessary concepts and scheduling operators in Rx are introduced.

Listing 3.8: Basic packages for user instantiations

```
generic
    with package Type_Traits is new Rx.Traits.Types (<>);
    --  The user has control via traits of definite storage for
    --  indefinite types, and conversions between them.
package Rx.Types is

    package Typed is new Rx.Impl.Typed (Type_Traits);
    -- This package encapsulates the Rx contracts and other
    -- supporting facilites. It is normally not needed unless
    -- the user wants to create new operators.

    package Observables is new Rx.Observables (Typed);
    -- This package contains sources and type-preserving operators.

end Rx.Types;
```

```
generic
   type T is private;
package Rx.Definites is
   -- This package contains default operator instantiations
   -- with sensible defaults for definite types for users
   -- without specific memory management needs.

   package Defaults is new Rx.Traits.Definite_Defaults (T);
   package Instance is new Rx.Types (Defaults.Type_Traits);
   -- Preparatory instances with defaults for definite types.

   package Observables renames Instance.Observables;
   -- Sources and type-preserving operators ready for use.
end Rx.Definites;
```

```
generic
   type T (<>) is private;
package Rx.Indefinites is
   -- This package follows a similar structure as Rx.Definites.
   -- Holders for indefinite types are used to wrap the user type.
   -- Those are used transparently for the user, which always deals
   -- directly with its indefinite type T (<>) in operators.

   -- Contents omitted.
end Rx.Indefinites;
```

```
generic
   with package From is new Rx.Observables (<>);
   with package Into is new Rx.Observables (<>);
package Rx.Operators is
   -- This package provides the type-transforming operators.
   -- The user instantiates it with the previous instances for
   -- individual types, obtained for example with the packages
   -- described above.

   -- Count, Flat_Map, Map, etc. declarations omitted here.
end Rx.Operators;
```

4.1. Thread Management and Schedulers

Rx uses the `Scheduler` abstraction to represent threads[3], thread pools and trampolines. A few tasking-specific operators use a scheduler parameter to switch the running thread, as seen in Listing 2.3. Per Rx specification, using such operators returns a particular task into which the execution flow will jump when an item is passed to the operator. This kind of single-thread management is useful, for example, in frameworks where blocking calls are forbidden in particular threads. (For example, in Android, blocking networking calls are forbidden in the main thread). It is, in general, a simple way of keeping the main task unblocked, akin to the use of futures.

Schedulers may encapsulate a single new thread, a specific existing thread, or even a thread pool (see Listing 4.1). Despite that, a key property is that the thread used by the thread-switching operators (see Listing 4.2) is guaranteed to be the same one for all items in a subscription. (Otherwise, that would force every operator to be thread-safe, which would be a breach of the Rx contract and an onerous distributed overhead.) In practice, the operators in Listing 4.2, `Observe_On` and `Subscribe_On`, achieve this property by picking the thread from the `Scheduler` within their `Subscribe` call, which occurs only once per operator.

[3] Since RxAda uses regular Ada tasks for Rx threads, these two terms are used interchangeably.

Listing 4.1: Standard Schedulers provided by RxAda

```
package Rx.Schedulers is

   type Thread is private;
   -- A wrapper for an actual Ada task.

   type Scheduler is tagged private;
   -- A wrapper for a thread pool.

   function Get_Thread (This : Scheduler) return Thread;
   -- Obtain a thread from this scheduler pool.

   -- Default schedulers are provided by Rx implementations.
   -- Custom schedulers can be created, e.g., to define thread
   -- pools with particular properties or allocation policies.
   -- Default schedulers in RxAda that are listed next return
   -- threads in round-robin order when Get_Thread is called.

   function Computation return Scheduler;
   -- Backed by a thread pool with one thread per CPU.

   function IO return Scheduler;
   -- Backed by an unbounded thread pool that returns an idle
   -- thread, when available, or a new one otherwise.

   function New_Thread return Scheduler;
   -- Always returns a new thread.

   function Immediate return Scheduler;
   -- Special scheduler that does not switch to a new thread.
   -- Code is executed in the same calling thread without delay.
   -- Scheduling future events with it is a bounded error.
```

Listing 4.2: Operators for task switching

```
generic
-- Formals omitted.
package Rx.Observables is

    function Observe_On (Scheduler : Rx.Schedulers.Scheduler)
                         return Operator'Class;
    -- This operator passes the item received during its own
    -- On_Next call to the thread obtained from the argument.
    -- Thus, the upstream operator preceding an Observe_On operator
    -- will see its On_Next call return immediately.
    -- The Scheduler thread, as soon as it becomes idle, will call
    -- On_Next in the operator following the Observe_On operator.
    -- On_Complete and On_Error calls are similarly relayed.

    function Subscribe_On (Scheduler : Rx.Schedulers.Scheduler)
                           return Operator'Class;
    -- This operator works in the opposite direction to Observe_On.
    -- Where Observe_On is useful during the Subscribed phase, and
    -- acts only on the On_* calls of the Rx contract, Subscribe_On
    -- acts during the Subscribe calls in the subscribing phase.
    -- When this operator Subscribe is called, it relays its call
    -- to the upstream operator Subscribe to the given thread.
    -- This way, the chain source operator thread can be selected.

    -- Other declarations omitted.
end Rx.Observables;
```

So, even if a thread pool scheduler is given to `Observe_On`, all items in the same subscription are processed by the same thread from the pool. Thus, by themselves, these operators do not enable parallel computation in the sense of, e.g., concurrent producers and consumers of data. For that, the `Flat_Map` operator is necessary as will be exemplified in Subsection 4.4. In preparation, the next subsection details RxAda `Thread` implementation, followed by the semantics and internals of the `Flat_Map` and related operators in Subsection 4.3.

4.2. Scheduler Implementation in RxAda

Schedulers are the basic Rx types used for thread creation and dispatching. To keep the RxAda implementation within the spirit of the Ada language, care has been taken to prevent deadlocking and manage termination of tasks. For example, in RxJava, when the main task exits, all other tasks are terminated too, which may cause pending events to go unprocessed, thus forcing the user to manually manage lifetimes. In RxAda, a single `Thread` is implemented by two Ada tasks. This is done in order to attain two objectives. Firstly, that no deadlock be possible because of convoluted use of `Observe_On`/`Subscribe_On` with the same Schedulers as arguments. Secondly, that tasks finish gracefully when all of them have an open `terminate` alternative. Thus, subscriptions in background threads will not be aborted prematurely, yet the user does not need to explicitly request to shutdown all pending tasks (although the `Rx.Schedulers.Shutdown` procedure exists for when premature aborting is wanted).

Fig. 3 details the implementation of an Rx thread and the responsibilities of each of its two tasks, which can be summarized as a queue manager and a code runner. On the left, the Queuer task awaits the arrival of events via its `Enqueue` entry. An event received while Idle can be immediately dispatched for execution, while events arriving while Busy will be queued in the priority queue of the Queuer task. Whenever an event is completed in the Runner task, Reap is accepted so the Queuer task can dispatch a new pending event (via the `Run` entry), or return to Idle state if the queue is empty.

Note that the Queuer is always ready to accept new events (at worst, after the bounded time of the `Enqueue` and `Reap` transitory states), even while Runner is executing an event. Hence, no deadlock is possible even if a Runner calls its own Queuer's `Enqueue` entry (e.g., through a downstream `Observe_On` with the same scheduler as argument). Finally, since both tasks block in accept statements that have `terminate` alternatives when Idle, this scheme allows clean, automatic shutdown managed by the Ada runtime.

4.3. Observers of multiple upstream subscriptions

Although the operators described in Subsection 4.1 allow switching the current thread in a subscription, by themselves they do not allow injecting concurrency within a chain, as there is only one flow from source to sink. To escape this limitation, some operators, like `Merge` or `Flat_Map`, can observe items coming from several observables at the same time. `Flat_Map`, furthermore, can create new secondary subscriptions in the context of another, main subscription. This capacity is the key to achieve concurrency.

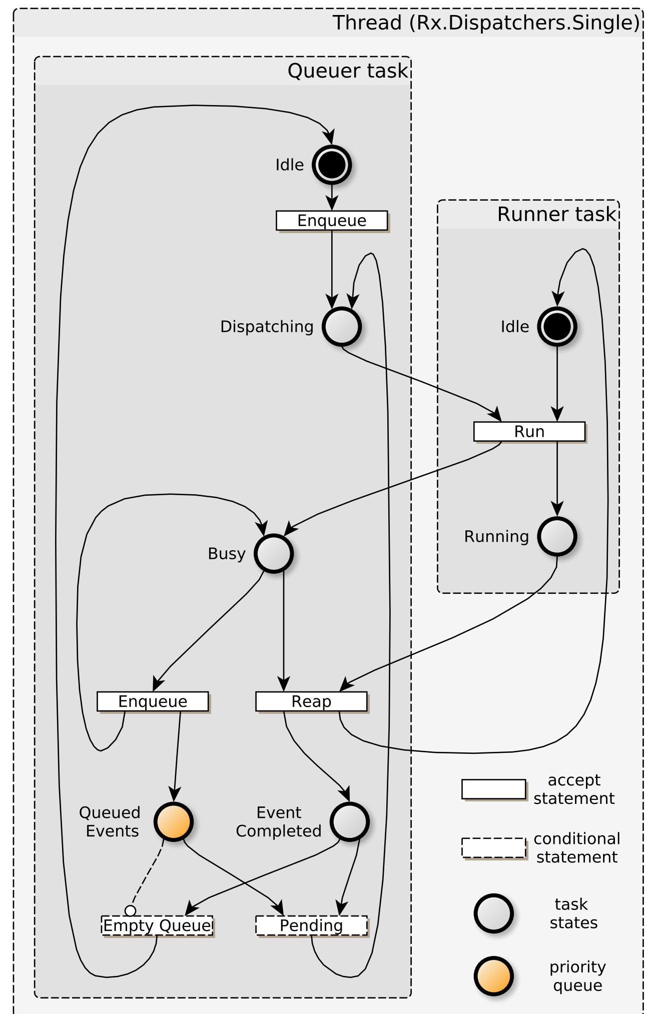

Figure 3: A single RxAda scheduler thread (implemented in `Rx.Dispatcher.Single`), backed by two Ada tasks. This Petri Net with inhibitor arcs describes the possible states and interactions among the tasks.

The confluence of several subscriptions in a same chain might seem to break the Rx promise of proper interleaving of items without overlapping, specially if concurrency can be created by `Flat_Map`. This subsection addresses the conundrum, explaining the internals of these operators in RxAda, and how they can achieve the Rx goal of providing concurrency in the context of the Rx contract.

The `Merge` operator, in its simplest signature, merges two observables' emissions into a single downstream chain. Naturally, both upstream observables must emit the same type of items. Generalizations of Merge take arrays of observables instead of a fixed number of them.

`Flat_Map` is possibly the most interesting and powerful operator in Rx, and also the most challenging one to Rx newcomers. The simplest signature of `Flat_Map` takes a function

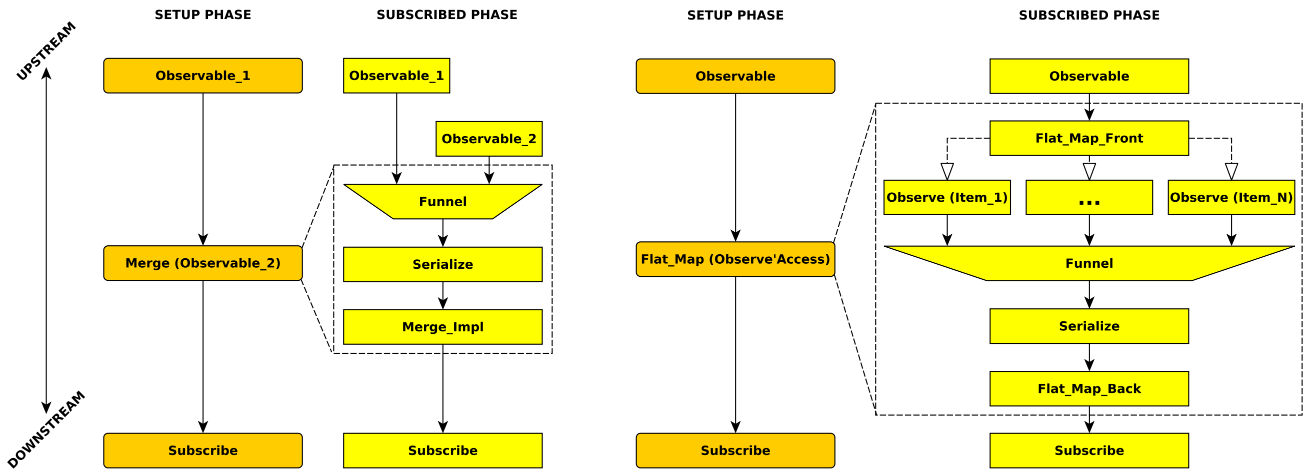

Figure 4: Inner design details of the `Merge` and `Flat_Map` operators. During subscription, the single operator seen by the user gets transparently replaced by the internal sequence of operators depicted in the respective subscribed phases.

argument that, given an item, returns a new observable derived from the item. When subscribed to, `Flat_Map` as expected subscribes itself to its upstream observable. We can refer to this subscription as the main, or primary subscription.

When items start to come in via the primary subscription, `Flat_Map` turns these items into observables with its argument function and, immediately, creates secondary subscriptions to the new observables, merging their emissions and pushing down their emitted items downstream. Inversely put, `Flat_Map` emissions are the merging of all the emissions generated by the subscriptions to the observables created from all items arriving from upstream via the primary subscription.

Equivalently, `Flat_Map` can take as argument a partial sequence of operators (without a source of items at the beginning, nor a sink consumer at the end). For each received item, `Flat_Map` turns it into an observable that emits only the item itself[4], and subscribes the partial sequence to the `Just (Item)` observable, again merging all new emissions for downstream.

Finally, `Expand` is a particular case of `Flat_Map` that not only emits the items from the newly generated observables, but it itself subscribes to those observables (which consequently must emit the same upstream incoming item type), allowing the use of recursivity in Rx streams.

The common trait of all `Flat_Map` variations is that secondary subscriptions are created per item received from the upstream observable. These new subscriptions present an opportunity to create or reuse other threads besides the one the primary subscription of `Flat_Map` is run on, by interspersing the scheduling operators described in the previous subsection in the chain argument given to `Flat_Map`.

Remembering that Rx is, by design and by default, lazy about thread creation, the programmer has to use the scheduling operators to achieve concurrency. Once these operators are introduced, three "sections" can be distinguished in the pipeline from the source to the sink (presuming, for simplicity, that scheduling operators appear only in the chain argument passed to `Flat_Map`):

1. From the source observable to the `Flat_Map` operator is a section that is executed in the thread of the primary subscription.
2. The secondary subscriptions, from their own source to the moment they are merged by `Flat_Map`, run on the thread determined by the use of an explicit scheduling operator (or the `Flat_Map` thread otherwise).
3. The merged emissions, down to the downstream subscriber, still run in their own threads, but must not overlap their calls to the remaining operators between `Flat_Map` and the downstream subscriber.

Given the third point, these operators face a situation that, if unaddressed, would break the Rx contract about non-overlapping calls to `On_Next`: the merged subscription emissions, which may come from different threads, have to be pushed down in mutual exclusion. This synchronization, in Ada, would typically be achieved with a protected type. However, there are at least two reasons to avoid this choice in the RxAda case. Firstly, user-supplied function calls can last an unknown time, and should consequently not be performed inside a protected subprogram. Furthermore, if items were emitted from within a protected subprogram, any call further down the chain to a blocking operation (for example in other user-supplied code) would result in a bounded error.

To avoid these risks, and to implement the basic `Merge` and `Flat_Map` operators, RxAda relies itself in other operators. With these final pieces in place, most others Rx operators can be implemented by combining existing ones, thus leveraging Rx own properties for simplicity.

These operators are named `Sequential` and `Funnel`. The `Sequential` operator applies a semaphore to the Rx contract

[4]This is precisely the purpose of the `"Just"` operator.

methods, guaranteeing that downstream operators will see items in mutual exclusion, as required by the contract. Using semaphores also gives user code complete freedom for its own locking mechanisms when accessing global data, as it is not run inside a protected operation.

The `Funnel` operator is the chosen way in RxAda to momentarily sidestep the library use of deep-copy semantics. As each chain is duplicated to get fresh operators upon subscription, the problem in turn is that `Flat_Map` would use a new copy of the downstream chain for every secondary subscription, thus breaking the operator semantics. The `Funnel` operator circumvents those copy semantics by storing its downstream operator by reference. This way, a chain that starts at a funnel will not be duplicated during subscription.

All these concepts come together in the implementation of the `Merge` and `Flat_Map` operators. Fig. 4 shows schematically their building blocks and its assemblage during a subscription. `Merge`, on the left, funnels together both input observables, ensuring with a `Serialize` operator that mutual exclusion is honored for downstream observers. The internal `Merge_Impl` operator ensures that `On_Complete` is called only once both upstream observables have completed.

`Flat_Map` uses the same funneling technique. However, `Flat_Map` relies on two, front and back, internal operators. `Front`, upstream, is in charge of calling the argument function that returns an observable per incoming item, and creating the secondary subscriptions, subscribing the funnel to these item observables. `Back`, downstream, is in charge of completing the subscription only when all generated item observables, plus the initial upstream source, have completed.

Finally, the internal back-side operators, `Merge_Impl` and `Flat_Map_Back`, being downwards of a `Serialize`, do not need to take measures about mutual exclusion for their Rx contract methods. Only the front-side `Flat_Map_Front` internal operator requires a critical section to ensure that creation and counting of secondary subscriptions is performed in mutual exclusion.

4.4. Examples of concurrency patterns

This section is devoted to exemplify how the Rx methodology allows expressing, with small changes and in a concise way, different typical concurrent producer-consumer configurations. The example application[5] consists in looking for all files found at any depth under a certain starting folder, and listing them with their computed SHA256 hash. To do so, the example relies on the `DirX.Observables` facilities: the function `DirX.Observables.Directory_Entries` returns an observable that emits all of a given folder entries, either recursively or not. A new user type, `Hashed_Entry`, stores a file name and its hash. Finally, the `Hash` function takes a `Directory_Entry`, computes its hash and returns a `Hashed_Entry`, so its profile is suitable for use with the `Map` operator.

In increasing grades of concurrency, the examples are: a baseline sequential 1-producer/1-consumer implementation; a 1-producer/N-consumer solution where hashes are computed concurrently; and an M-producer/N-consumer where both directory listing and hash computation are done concurrently. See the Rx chains in Listings 4.3 to 4.5, in Fig. 5. With minimal modifications and using standard Rx operators, these examples could be retailored for other purposes: timing can be achieved with the `Stopwatch` operator; deterministic cumulative hashing by adding sorting and string hashing operators; counting with the `count` operator; and so on.

For the 1-1 and 1-N cases, the file enumeration is done internally and recursively in the caller task by the `Directory_Entries` source observable. Concurrency for the hashing is achieved with the `Flat_Map` operator: the secondary operator chain that it takes as parameter is subscribed to every incoming `Directory_Entry` item, hence using a different thread from the computation pool. `Flat_Map` then funnels all resulting `Hashed_Entry` items downstream, in mutual exclusion, for final printout.

For the M-N case, the `Expand` operator must be used explicitly to enumerate folder contents in the `Input_Output` pool. Regular files are pushed down unmodified, and directories are expanded into their contents. Since `Expand` is the recursive specialization of `Flat_Map`, its chain argument is also subscribed to every new item, allowing the introduction of concurrency (in this case, producing entries in an input/output pool for subsequent hashing).

The reader is invited to reflect on a classic Ada solution, that would involve the definition of worker threads, protected objects for queuing of jobs, the task pool, management of two job types (folder enumeration and file hashing), etc. Looking forward to upcoming Ada 202X parallelism [13], a reduction in the amount of boilerplate in regular Ada is to be expected: parallel iteration over the elements of a collection will no longer require protected sources nor explicitly defining tasks, for example. Comparisons of syntax and performance between RxAda and Ada 202X, once the latter is finalized, will thus be of interest.

5. Library Organization

The RxAda library has been structured in several package families to simplify its maintenance and understanding. For example, the `Rx.Impl.*` hierarchy contains packages that would rarely be of interest for final users, and that contain most of the logic implementing the Rx framework. Other similar branches are presented next, concluding with the multithreading implementation support packages.

5.1. User-Facing Packages

An effort has been made to isolate as much as possible packages that users of the library may want to eventually know about from other implementation packages. This separation is visible in two ways: in the on-disk file organization and in the package names. Source files (available at [10]) are classified in three folders. A first, root folder named `"src"` contains user-facing packages, like the ones discussed in Section 3.5. Within this folder, another one

[5]The full code can be found at https://github.com/mosteo/rxada/blob/jsa2019/dirx/src/examples/dirx-hash_recursive.adb

Listing 4.3: 1-1 sequential solution

```
Subscription :=
   Directory_Entries (Target,
                      Recursive => True)
     -- Emits all entries under the given
     -- Target folder. Folders are entered
     -- recursively as found, until all the
     -- subtree is emitted.

   & Map (Hash'Access)
     -- Hashes a file contents with SHA256.
     -- Folders are let through with an
     -- empty hash (other options could be
     -- considered like hashing its entry
     -- names, or hashing its entries
     -- hashes. This latter option could be
     -- achieved without rehashing by using
     -- the Group_By operator, for example.

   & Subscribe (Print_Hash'Access);
     -- Final printing of name and hash.
```

Listing 4.4: 1-N concurrent hashing

```
Subscription :=
   Directory_Entries (Target,
                      Recursive => True)
     -- Same as in the 1-1 case.

   & Flat_Map (Observe_On (Computation)
               & Map (Hash'Access))
     -- Flat_Map with an operator chain as
     -- argument converts each incoming
     -- item to a Just (Item) observable,
     -- to which the argument chain is
     -- subscribed. Thus, each item is
     -- observed by Map in a Round-Robin-
     -- picked thread from the Computation
     -- pool. Alternatively, Map could be
     -- followed by a Subscribe_On
     -- operator instead.

   & Subscribe (Print_Hash'Access);
     -- Same as in the 1-1 case.
```

Listing 4.5: M-N enumeration & hashing

```
Subscription :=
   Directory_Entries (Target,
                      Recursive => False)
     -- Recursivity is disabled to take
     -- advantage of Expand, below.

   & Expand (Observe_On (Input_Output)
             & DirX.Observe'Access)
     -- DirX.Observe takes a Dir_Entry and
     -- returns an Observable that emits
     -- all folder contents as Dir_Entry.
     -- By preceding it with Observe_On,
     -- the enumeration takes place in an
     -- Input_Output pool thread.

   & Flat_Map (Observe_On (Computation)
               & Map (Hash'Access))
     -- Same as in the 1-N case.

   & Subscribe (Print_Hash'Access);
```

Figure 5: Three RxAda implementations of a directory tree contents hashing. Some package prefixes are omitted for conciseness.

named `"priv"` contains the specifications of implementation packages. Finally, a sibling folder `"body"` contains all bodies of the library, which should not be needed by users, but that are this way easier to find for maintainers and contributors. In summary, basic users should concern themselves with the `"src"` folder whereas advanced users might have interest at some point in the `"priv"` folder too. Packages in the `"src"` folder are named as `Rx.<Name>`, while implementation packages follow a `Rx.<Specialization>.<Name>` convention, as detailed next.

5.2. General Implementation Packages

As seen, implementation packages without a more specific classification belong in the `Rx.Impl.*` hierarchy. These packages deal with Rx concepts, contrarily to other non-Rx-specific supporting packages (that might as well be provided by third-party libraries, although there are no external dependencies at present) that are found in `Rx.Tools.*`. The supporting packages include, for example, semaphores and thread-safe reference-counting pointers used for example in the implementation of the `Funnel` and `Serialize` operators seen in Section 4.3. The rest of operators rely on regular by-value copy semantics during chain concatenation, and reference types during data flow, simplifying the library memory management.

5.3. Operator Implementations

In order to distinguish between observables that create items (sources) and operators that transform data, these packages are classified respectively under the `Rx.Src.*` and the `Rx.Op.*` hierarchies, although (non-Ada) Rx documentation usually refers to them indistinctly as operators.

5.4. Scheduling Packages

As seen in the examples presented through this work, and particularly in Section 4.4, Rx allows simple yet powerful task management. As in other Rx implementations, users need typically to use the `Rx.Schedulers` package facilities for task control, unless custom pools are wanted. The actual implementation of tasking events and pools that are used in `Rx.Schedulers` and the `Subscribe_On`, `Observe_On` operators is filed under the `Rx.Dispatchers.*` branch. An abstract `Dispatcher` interface is defined that allows scheduling events in a particular scheduler at a particular time; this is used to implement the different specialized task pools recommended in the Rx documentation (e.g., I/O and background computation), and would be the starting point for the creation of custom user pools. These packages contain, for example, the implementation detail corresponding to Fig. 3.

6. Conclusion

This work presented an Ada 2012 implementation of the ReactiveX approach to reactive programming. The focus was placed on design and implementation decisions adopted to address the challenges arising from the imperative nature of Ada, tagged types syntax, lack of implicit instantiations of generics and lambda functions, and explicit memory management. Concurrency design and implementation details were explored, since multithreading management is one of the strong values of adopting Rx. Examples of various producer-consumer configurations were provided to illustrate the tasking flexibility of Rx, and its RxAda implementation, through a file-hashing application.

The implementation of the library aims at balancing user comfort, maintenance simplicity, and correctness and performance issues such as compile-time type checking. To that end, the library is structured in two main layers: the upper layer exposes the operators to users with purely client needs in a type-centric manner, with a single generic instantiation required per type and per transformation between types. The

lower layer contains features needed for library expansion and advanced client use like the definition of new operators or selective instantiation of partial subsets of the library.

RxAda is available under an Open Source license to interested parties.

References

[1] Bainomugisha, E., Carreton, A.L., Cutsem, T.v., Mostinckx, S., Meuter, W.d., 2013. A survey on reactive programming. ACM Comput. Surv. 45, 52:1–52:34.

[2] Bonér, J., Farley, D., Kuhn, R., Thompson, M., 2014. The Reactive Manifesto. URL: http://www.reactivemanifesto.org/.

[3] Briot, E., 2015. Traits-based containers. URL: http://blog.adacore.com/traits-based-containers.

[4] Edwards, J., 2009. Coherent reaction, in: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA. pp. 925–932.

[5] Elliott, C.M., 2009. Push-pull functional reactive programming, in: Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell, ACM. pp. 25–36.

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman, Inc.

[7] Maglie, A., 2016. ReactiveX and RxJava, in: Reactive Java Programming. Springer, pp. 1–9.

[8] Meijer, E., 2010. Subject/observer is dual to iterator, in: FIT: Fun Ideas and Thoughts at the Conference on Programming Language Design and Implementation.

[9] Meijer, E., 2012. Your mouse is a database. Commun. ACM 55, 66–73.

[10] Mosteo, A.R., 2017a. RxAda. URL: https://github.com/mosteo/rxada.

[11] Mosteo, A.R., 2017b. RxAda: An Ada implementation of the ReactiveX API, in: Ada-Europe International Conference on Reliable Software Technologies, Springer. pp. 153–166.

[12] Salvaneschi, G., Margara, A., Tamburrelli, G., 2015. Reactive programming: A walkthrough, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 953–954. doi:10.1109/ICSE.2015.303.

[13] Taft, T., 2019. Ada 202X lightweight parallelism and OpenMP. Ada Rapporteur Group.

[14] The Rx Team, a. ReactiveX: An API for asynchronous programming with observable streams. URL: http://reactivex.io/.

[15] The Rx Team, 2010b. Rx design guidelines. URL: https://blogs.msdn.microsoft.com/rxteam/2010/10/28/rx-design-guidelines/.

[16] Thornley, J., 1993. Parallel Programming with Declarative Ada. Technical Report. Caltech.

[17] Wan, Z., Hudak, P., 2000. Functional reactive programming from first principles, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM, New York, USA. pp. 242–252.