WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRONICS

FIELD:	Control Engineering and Robotics (AiR)
SPECIALIZATION:	Embedded Robotics (AER)

MASTER OF SCIENCE THESIS

Reactive functional programming in distributed embedded systems

Reaktywne programowanie funkcyjne w rozproszonych systemach wbudowanych

AUTHOR:
Tomasz Bartos

SUPERVISOR:
Mariusz Janiak, Ph.D.

GRADE:

WROCŁAW 2018

Abstract

Consideration included in this thesis reflect on application of reactive and functional programming paradigms in distributed embedded systems. Application of presented paradigms is related to methods of software components design, which are based on processing of discretely occurring events in time. Processing of asynchronous event streams has been realized in accordance with declarative programming approach. In proposed model, relations between streams have been exposed by the set of reactive operators, which allow for efficient and safe operating on the data in multi-thread environments. Discrete streams of events have been modelled by distributed communication protocol, which has application to existing robotic control systems. Both concepts have been merged and presented in a developed library, which has been used in performed simulations. Correctness of proposed approach has been verified by creation of exemplary robotic application, which is responsible for realization of monitoring tasks in accordance with reactive and functional programming paradigms.

Streszczenie

Rozważania zawarte w pracy dotyczą zastosowania paradygmatów reaktywnego i funkcyjnego programowania w rozproszonych systemach sterowania. Zastosowanie paradygmatów odnosi się do sposobów projektowania oprogramowania po stronie aplikacji, które są oparte na przetwarzaniu dyskretnych zdarzeń występujących w czasie. Przetwarzanie asynchronicznych strumieni zdarzeń zostało zrealizowane w sposób deklaratywny. W przedstawionym modelu, relacje pomiędzy strumieniami zostały wyrażone za pomocą zbioru reaktywnych operatorów, które pozwalają na wydajne oraz bezpieczne operowanie na danych w środowiskach wielowątkowych. Dyskretne strumienie zdarzeń zostały zasymulowane przez zastosowanie rozproszonego protokołu komunikacyjnego, który znajduje zastosowanie w istniejących robotycznych systemach sterowania. Obie idee zostały połączone i zaprezentowane w stworzonej biblioteki, która została wykorzystana podczas wykonywanych symulacji. Prawidłowość zaproponowanego podejścia została zweryfikowana poprzez stworzenie przykładowej aplikacji robotycznej, realizującej zadania w zgodzie z reaktywnym i funkcyjnym podejściem programowania.

Contents

List of Acronyms

CBD *Component-based development*[7][8]. 10, 37, 39

CPU *Central processing unit*. 9

DDS *Data Distribution Service*[44]. 38, 39

EDP *Endpoint Discovery Protocol*[46]. 38

FP *Functional programming*[19]. 22, 27, 41

FRP *Functional Reactive Programming*[15]. 20–22, 41, 66

IDL *Interface Description Language*[47]. 38

OOP *Object-Oriented Programming*[10]. 14, 22

P2P *Peer-to-peer*[50]. 39

PDP *Participant Discovery Protocol*[46]. 38

Pub/Sub *Publish-subscribe pattern*[43]. 37–39

QOS *Quality of Service*. 38

RAII *Resource Acquisition Is Initialization*[58]. 43

ROS *Robot Operating System*[51]. 39, 41–43, 47, 54, 59, 75

RP *Reactive Programming*[30]. 26, 27, 41, 43

RPC *Remote Procedure Call*[53]. 39

RTPS *Real Time Publish Subscribe*[46]. 38, 41

Rx *ReactiveX*[55][56]. 41, 43, 44, 57

SCT *System Component Tests*. 36

SIT *System Integration Tests*. 36

SOA *Service Oriented Architecture*[41]. 37

SRP *Single Responsibility Principle*[42]. 37

UDP *User Datagram Protocol*[49]. 38

UML *Unified Modelling Language*[14]. 18, 19, 42, 73, 74

Chapter 1

Introduction

Origins of the software development were associated with the appearance of the first transistor based *Central processing unit* (CPU), which successfully displaced solutions based on unreliable relays and vacuum tubes. In 1965, Gordon Moore made an observation, that the number of transistors inside processor would be doubled every two years[1]. On the basis of this assumption, software development over the years was based on operations performed on the single cores of the processors. The trend of the line shown in Figure 1.1 was surprisingly accurate during next 50 years after publication. The author predicted, that reliance would reach the end at the first quarter of 21st century[2]. Described fact changed the line of thought about structure of the systems and induced developers to make use of the multi-thread scalability.

These days, parallel computations are visible in getting more popular *cloud* computing solutions. In this model, computations are performed on distributed machines, which are composed of a loosely coupled clusters. This approach gives the possibility to reduce summarizing cost of calculations, because computations are performed outside users device. On the other side, the presented concept allows to obtain higher scalability and reliability of the systems, which are achieved by a replication of the services.

In similar way, significant impact on development of parallel and distributed systems has increasing number of more and more demanding users. In 2015 Sandvine organization published statistics presented in Figure 1.2. With reference to the results, more than 37% of downstream peak traffic in North America was dominated by Netflix company. Seeing that the most important are numbers, because Netflix pushed $329,400,000,000$ gigabytes of data by video streaming to around 60 mil-

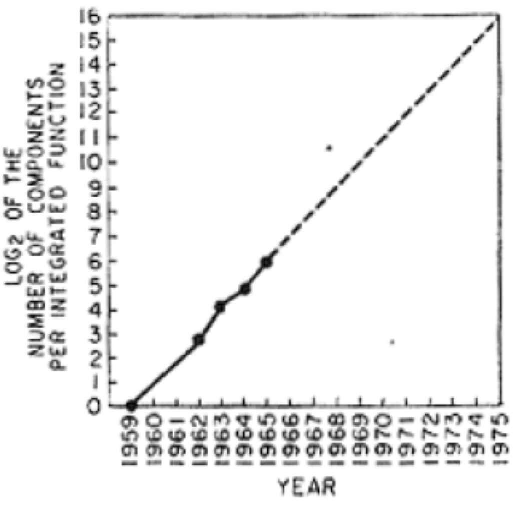
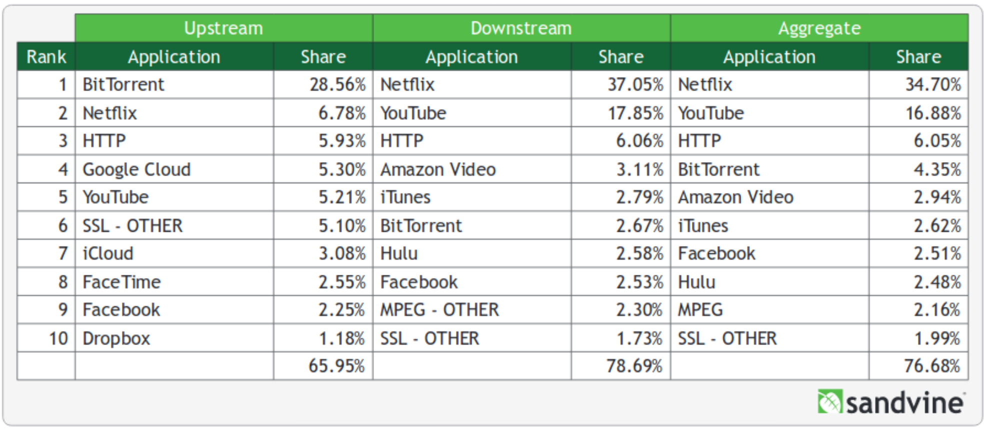


Figure 1.1 Moore's law trend line proposed in 1965[1].

lions of subscribers[3]. As can be seen, there has been solved a huge asynchronous problem. Each connected client should be served independently preserving a high quality of provided services. Increasing users demands, for reliable and low latency services, have been solved by decomposition of systems into smaller sub-services communicating with each other[4]. What is more, on the level of data processing have been applied techniques characteristic for reactive and functional programming paradigms, so that external world model has been expressed by sequentially occurring events in time[5].

Similar requirements for reliable processing can be observed for robotic systems, because short time of reaction and stability of work influence on health and safety of humans life. For that reason, for robotic systems have been defined software architectures, which allow to eliminate most of the described problems. One of the most popular approaches is *Component-based development*[7][8] (CBD), wherein the logic of the robotic system is separated into independent applications. In this structure, each component is responsible for the realization of predefined logic and communication with other parts of the system. In particular case, logic separation allows for ordering of responsibilities, so that each domain can realize well-defined tasks and stay focused on correctness and time limitations of performed calculations. In order to meet the requirements, most of the computations must be performed in parallel, which formulates a need to define methods of efficient and safe asynchronous data processing.



	Upstream		Downstream		Aggregate	
Rank	Application	Share	Application	Share	Application	Share
1	BitTorrent	28.56%	Netflix	37.05%	Netflix	34.70%
2	Netflix	6.78%	YouTube	17.85%	YouTube	16.88%
3	HTTP	5.93%	HTTP	6.06%	HTTP	6.05%
4	Google Cloud	5.30%	Amazon Video	3.11%	BitTorrent	4.35%
5	YouTube	5.21%	iTunes	2.79%	Amazon Video	2.94%
6	SSL - OTHER	5.10%	BitTorrent	2.67%	iTunes	2.62%
7	iCloud	3.08%	Hulu	2.58%	Facebook	2.51%
8	FaceTime	2.55%	Facebook	2.53%	Hulu	2.48%
9	Facebook	2.25%	MPEG - OTHER	2.30%	MPEG	2.16%
10	Dropbox	1.18%	SSL - OTHER	1.73%	SSL - OTHER	1.99%
		65.95%		78.69%		76.68%

sandvine

Figure 1.2 Top 10 Peak Period Applications - North America, Fixed Access[6].

1.1 Purpose and scope of the work

In this thesis will be proposed method of components code design based on reactive and functional programming paradigms. There will be verified, if selected properties of described paradigms find application in distributed robotic control systems and will simplify process of asynchronous data processing. For that reason, as technical aspect of this thesis has been developed library, which is responsible for composition of defined paradigms with existing robotic solutions. In order to verify correctness of proposed approach, on the basis of developed framework, exemplary robotic application has been developed.

Author will emphasize, that entire software has been developed only with usage of open-source resources.

Chapter 2

Selected aspects of software engineering

In this chapter will be introduced selected theoretical aspects of software engineering. Considerations will start from general classification of selected programming paradigms and software design patterns. Next, there will be introduced theoretical aspects of declarative programming paradigms with description of their most important properties. Described concepts will be used as a foundation for developed library (Chapter 4) and exemplary robotic application (Chapter 5).

2.1 Common programming paradigms

Solving of the complex programming problems requires choosing of the right approach. For that reason have been created different programming languages, which are supporting one or many programming paradigms. In next sections will be shown considerations related to most general representatives.

2.1.1 Imperative programming

The concept of imperative programming paradigm appeared together with the occurrence of first high-level programming languages[9] in the 1950s. The general idea of this model was originally inspired by the structure of computers memory,

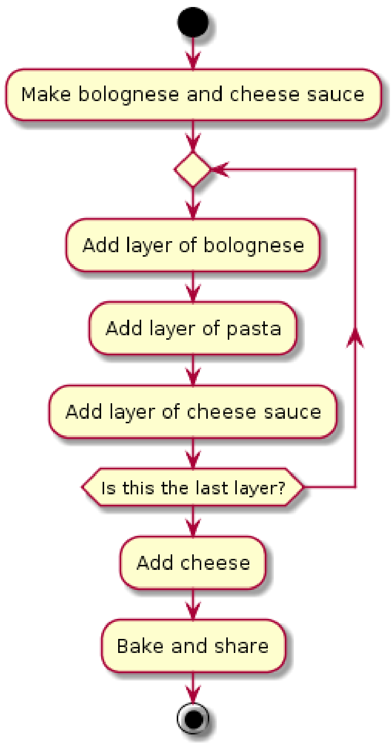


Figure 2.1 Flowchart with algorithm of lasagne preparation.

which is a set of associated values stored in different locations[10]. Programs during the realization of its logic were mainly focused on operating on those states with a usage of assign statements.

On the conceptual side, imperative paradigm is describing “how” to achieve some result. Flow of application is mostly realized by instructions, which sequentially modify state of the program. Set of statements can be understood as procedure or algorithm. One of the most popular representatives of imperative paradigm is *Object-Oriented Programming*[10] (OOP). In OOP structure of applications is divided into composable blocks. An exemplary step-by-step recipe containing procedure of lasagne preparation has been shown in Figure 2.1.

2.1.2 Declarative programming

Declarative paradigm describes “what needs” or “what is” to be computed. In other words, application logic is expressed without its control flow description.

Programs are mostly composed of declarations, which describe types and relations between them. In contrast to the imperative example of lasagne preparation shown in Figure 2.1, declarative conceptual definition of lasagne can be defined as follows:

- "*Lasagne* is grated cheese on cheese sauce on flat pasta on cheese sauce on Bolognese on flat pasta on cheese sauce on Bolognese on flat pasta on cheese sauce baked for 45 minutes.
- *Bolognese* is onion and oil fired until golden mixed with ground beef mixed with tomato simmered for 20 minutes.
- *Cheese sauce* is milk and cheese added progressively to roux while frying it until the sauce thickens.
- *Roux* is flour and butter fried briefly.
- *Baked* is put in oven dish in a hot oven.
- *Fried* is put in a pan on high and mixed frequently.
- *Simmering* is put in pan on low and mixed infrequently."[11]

2.2 Software design patterns

During long-standing process of software engineering, developers noticed that they had been solving many of engineering problems in similar way. Presented observation initiated concept of programming patterns creation, wherein four famous architects known as "Gang of four" extracted most popular patterns and described them in [12]. Formulation of patterns allows to define conventions and extract common building blocks, which can be used as an informal language in describing the parts of the code. These patterns may be useful, because humans are perfect machines created for patterns recognition[13].

Patterns described by the authors can be organized into groups, on the basis of the problems which they solve. The following categories can be distinguished:

- **Creational** - describes the initialisation process of objects, in order to separate user from details about structures creation, composition or internal representation.

- **Structural** - describes how objects can be composed to form larger structures.
- **Behavioral** - describes how can be represented communication and responsibilities between objects.

Diagram with relationships between design patterns has been shown in Figure 2.2. In the terms of the following thesis, in further sections will be described design patterns, which are necessary from perspective of further considerations.

2.2.1 Iterator pattern

One of the representatives of behavioural patterns is called *iterator* (informally called *cursor*). Idea of iterator is based on an abstraction, which allows for traversing through elements of data structures, without exposition of their implementation details. On the iterator abstraction level, user is able to ask for next elements of the collection, without knowledge where they are physically located. Access to elements can be performed sequentially in different order, depending on which type of iterator has been chosen. Exemplary types of iterators have been introduced below:

- **Forward iterator** provides access to elements from begin to end of collection.
- **Reverse iterator** provides access to elements in reverse direction from end to begin of collection.
- **Random access iterator** allows to access elements sequentially in random order without duplication of visited places.

Class diagram of the pattern has been shown in Figure 2.3. As can be observed, iterator contains following informations about iterated structure:

1. Address to the first element.
2. Address to currently visited element.
3. Information about existence of the next element.

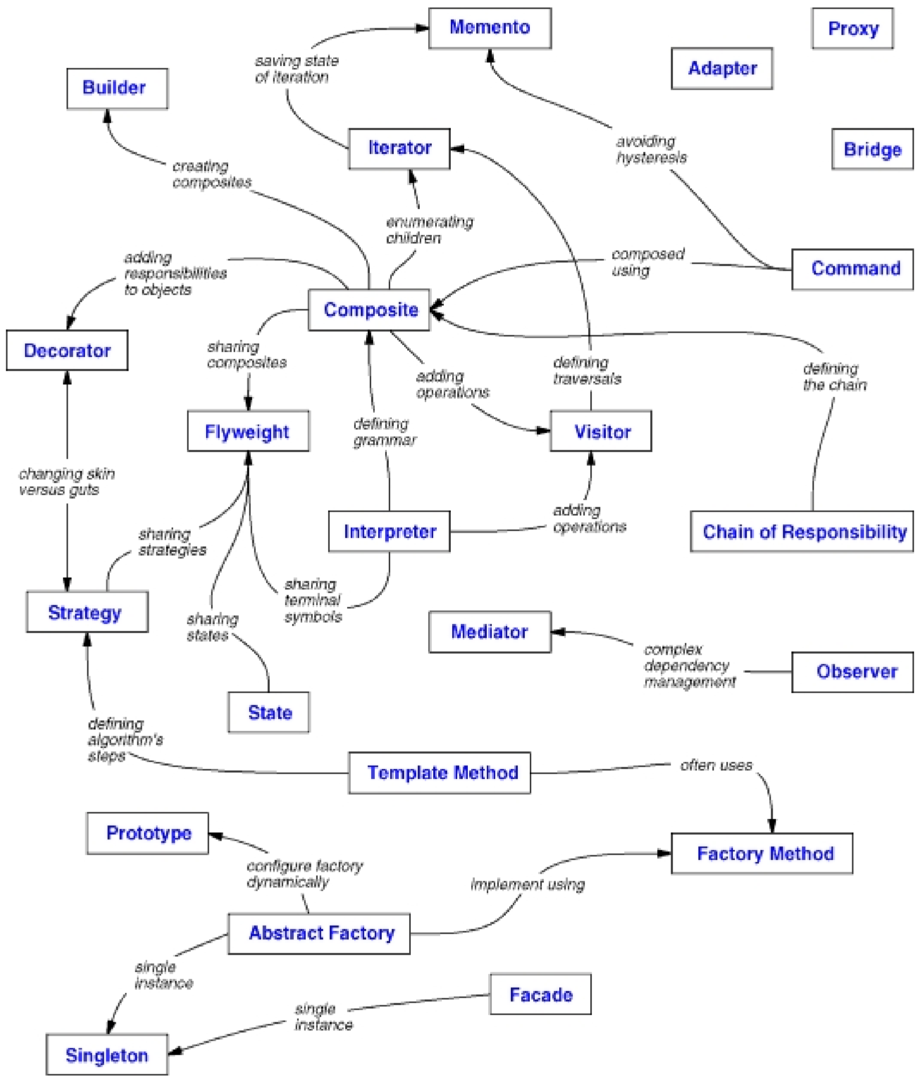


Figure 2.2 Relationships between design patterns[12].

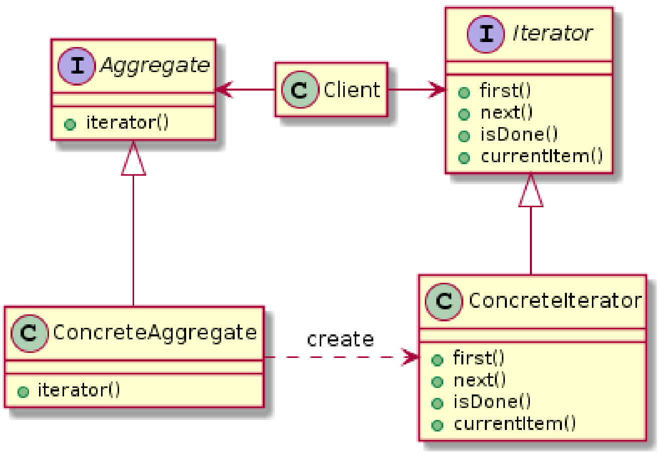


Figure 2.3 Class diagram for iterator pattern in UML notation.

4. Address of the next element (if exists).

Iterator pattern can be also applied for generators, which are not storing elements in memory but are generating new values basing on the previously known state. Generators differ from data structures because the number of returned elements can be infinite[1].

2.2.2 Observer pattern

Another popular representative of behavioural patterns is called *observer*. The main motivation in construction of this pattern is to loose couple between some source of the data and potential subscribers interested on it.

Structure of the pattern has been shown in Figure 2.4. Observer pattern realizes one to many relation, which allows to store multiple observers inside *subject*. Concept of interactions between classes is mainly based on *Subject* class, which is responsible for observers notification, when any change of observed object occurs. Observable change can be signalized by call of *notifySubscribers* method.

In the Figure 2.5 has been visualized exemplary behaviour of observer pattern. In particular situation, structure of the pattern has been composed of single source

[1]Storing of infinite number of elements in data structures is impossible due physical memory limitations.

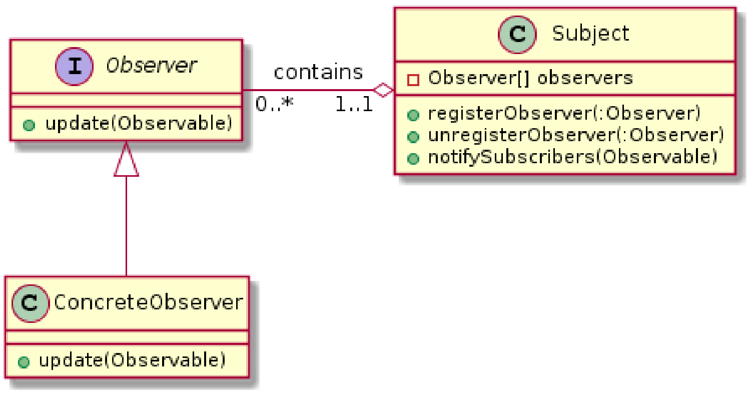


Figure 2.4 Class diagram for observer pattern in UML notation.

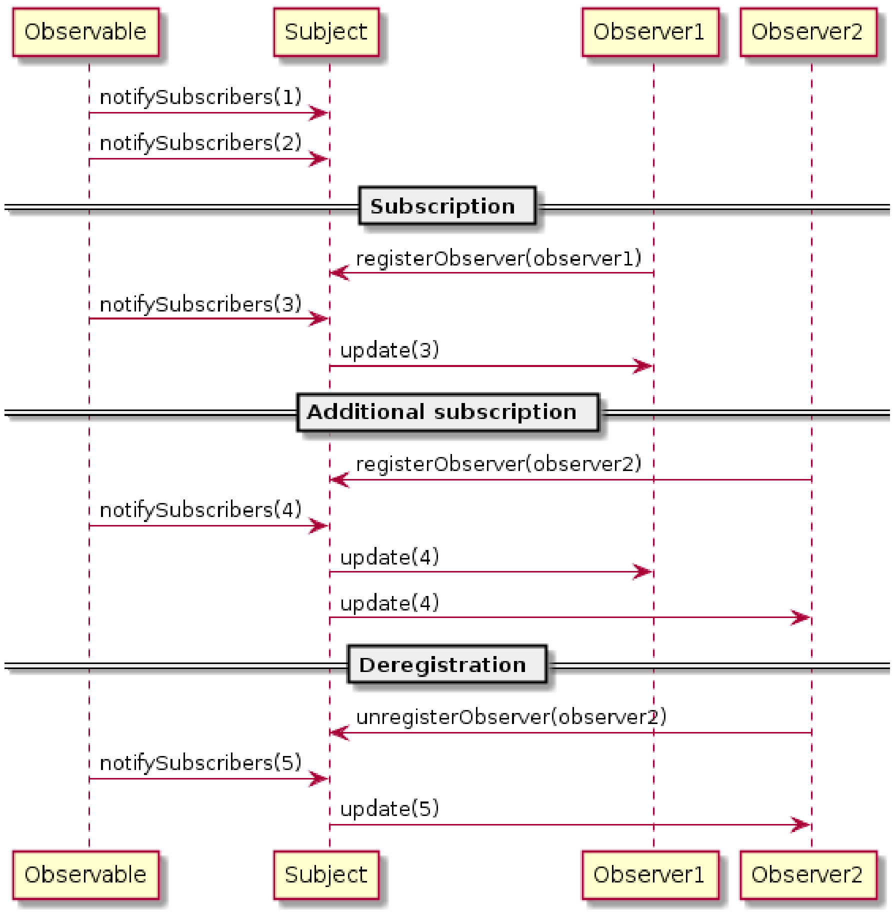


Figure 2.5 Sequence diagram representing flow of data for observer pattern.

generating next natural numbers and two observers interested in this data. In the first part of processing subscribers are not registered. In this moment every information received by the subject is skipped. If any of observers decide on subscription, then it can signalize it by call of *registerObserver* method. Since this moment, observer is notified about every integer passed to the subject. Each subscription can be treat independently, according to assumptions defined at the beginning of this section. Attachment of additional subscriber has been shown in next section of the diagram. As can be seen for next notification, both subscribers have been notified simultaneously. From the observer point of view, information about source of the data and existence of other subscribers are unknown. Finally, subscriber can decide to de-register subscription and interrupt notification process. Detachment does not impact other subscribers.

2.3 Functional Reactive Programming

First official paper describing concept of *Functional Reactive Programming*[15] (FRP) was published in 1997 by Conal Elliott and Paul Hudak. Given paradigm was defined in order to provide formal and declarative description for time-varying values. Time-varying values can be understood as a different concept for modelling values changing over time[2]. FRP introduces two types called *behaviours* and *events*.

2.3.1 Behaviour

The meaning of behaviour can be understood as a model for representing continuous flow of values. Formally, as shown in Equation 2.1, behaviour has been defined as a function, which for any real point in time T assigns discrete value α. From its nature, behaviour assumes that value of α always exists and can continuously vary over time. In particular case, continuousness is related to model of time, which is represented by real and non-decreasing values.

$$\mu :: Behaviour\ \alpha \rightarrow (T \rightarrow \alpha), \quad where\ T = \mathbb{R} \tag{2.1}$$

As an example of behaviour can be proposed temperature of processor, which is

[2]Similar model was partially introduced in Section 2.2.2.

continuous[3] by its nature. Temperature also fulfils requirement concerning existence of the value, because there does not exist value of T for which behaviour will be unable to assign value of the temperature.

2.3.2 Events

In contrast to behaviour model, events are used to describe values at a particular moments in time. Basing on this assumption, lifetime of event is associated with single time point, which as opposed to behaviours is discrete. In other words, event can be interpreted as a pair composed from time T and a value α, which has been formally described in Equation 2.2.

$$\mu :: \mathit{Event}\ \alpha \rightarrow [(T,\ \alpha)], \quad \mathit{where}\ T\ \mathit{is\ not\ decreasing} \tag{2.2}$$

Single occurrence of event is not often concerned, instead of it are composed sequences of events known as streams. Stream can be modelled as infinite list of pairs ordered by time, containing both discrete moment of event occurrence and its value. Exemplary can be considered input from the keyboard, that each press of the button generates event and exists only when the button is pressed.

2.3.3 Original formulation concepts

Mathematical concept of FRP is based on two fundamental principles:

- Continuous time model.
- Denotational semantics.

Continuous time model is strictly related to concept of behaviours, which can continuously vary over time. Realization of continuous time model on discreetly clocked computers may be not obvious, because behaviour of the processor is discrete even if really high clocking frequency is used. For that reason, behaviours should be understood as functions that always model continuous nature, which does not prevent us from querying them in any discrete point in time[16].

[3]For every two points in time, there exists infinite number of other points between them.

Denotational semantics[17] in particular case can be understood as implementation independent approach, that formally specifies the type system and its building blocks with usage of mathematical objects (denotations). In reference to FRP, given specification allows for formal specification, that compositional nature[18] defines the correctness for all combinations of building blocks, which are consistent with type system. One of the reasons of given proof is that formal definition separates specification from programming implementation details. The second one is that composable semantic model allows for formal definition of operators called *combinators*, which can be used to operate on events and behaviours, in order to define their relations and express expected application logic[4].

2.4 Functional programming

Functional programming[19] (FP) can be understood as declarative paradigm focused on avoidance of mutable data, shared state and side-effects. In a contrast to OOP, instead of statements execution, FP is based on the evaluation of expressions[20][5]. FP due its properties has application to multi-thread environments, because it allows to efficient and more predictable parallel programming. In next sections will be introduced properties of FP necessary from perspective of further considerations.

2.4.1 Shared and immutable state

In computer science, shared state can be understood as any resource, memory space or object, which is shared between two or more functions or data-structures. Shared states can be used to exchange informations between different parts of the system. From perspective of shared resource, its value can vary over time or not. For that reason, states can be divided into constant (immutable) and mutable groups. Presented contradistinction is important from perspective of memory model in programming languages[6], because not synchronized access to mutable and shared resource can result in data races and race conditions[23].

According to synchronization quadrant shown in Figure 2.6 and previous explana-

[4]Concept of operators will be described in more details in Section 2.5.4.

[5]FP is based on *lambda calculus*[21] computational model (formally equivalent with *Turing machine* concept[21]), which allows for expression of each state based program with functional

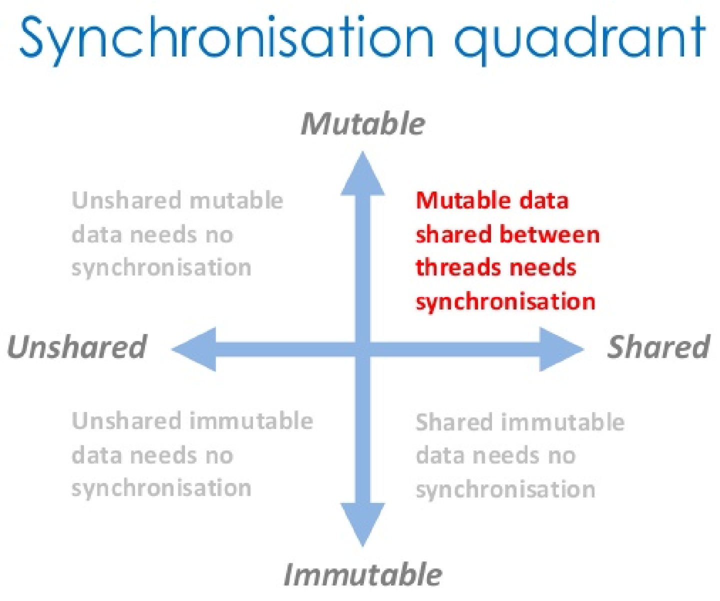

Synchronisation quadrant
Mutable
Unshared mutable data needs no synchronisation
Mutable data shared between threads needs synchronisation
Unshared
Shared
Unshared immutable data needs no synchronisation
Shared immutable data needs no synchronisation
Immutable

Figure 2.6 Diagram representing when access to resource must be synchronized[24].

tions, access to resource must be synchronized only when value is shared and its state can be modified[7].

Functional programming, by avoidance of shared state, solves problem with simultaneous access to mutable data, because relaxation of any from given assumptions allows for unrestricted and parallel access to the resource. For that reason, functional processing is based on concept of immutability, that each created resource can not be modified since its creation. If there appears a need of state modification, then mostly it will be realized by creation of new immutable state on the basis of existing one (copy).

paradigm.

[6]In particular case, under term of programming language will be understood C++11 memory model[22], with focus on sections 1.7 and 1.10.

[7]Modifications performed on shared and mutable values are characteristic for imperative approach and require additional synchronization mechanisms.

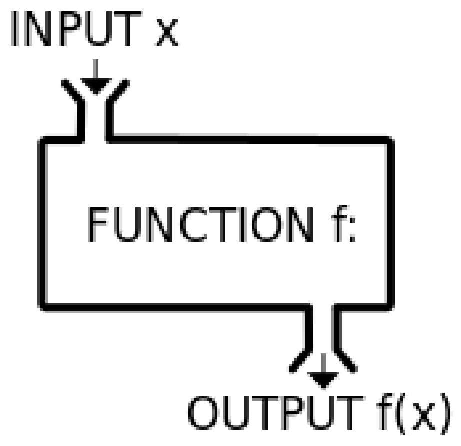


Figure 2.7 Pure function concept[25].

2.4.2 Pure function

Pure function from definition is a function, which for given set of arguments, always returns exactly the same results. This property can be achieved by avoidance of side effects, which are state changes visible outside function call and differ from its return value. Concept of pure function has been visualized in Figure 2.7, which represents function evaluation without external resources.

As an example of side-effect can be understood any network operation or random number generation, which are strongly related to particular moment in time or state of external resources. External dependency removes waranty about deterministic behaviour of function calls. For that reason, in pure functional languages like Haskell, there can be observed separation between side-effects and pure functions realized mostly with usage of a monad[26] concept.

$$f(x, y) = x \times y \tag{2.3}$$

One of the most important properties of pure functions is known as *referential transparency.* In Equation 2.3 has been defined function, which for any given two numbers returns its multiplication. Exemplary call of $f(6, 7)$ is in deterministic way always evaluated to 42 value[8], which means that function call can be replaced

[8]Answer to the "Ultimate Question of Life, The Universe, and Everything" made by "Deep

Listing 2.1 General explanation of UNIX pipeline functionality.

```
process1 | process2 | process3
```

Listing 2.2 Data processing based on UNIX pipeline functionality.

```
echo "The future starts today, not tomorrow." | cut -d ',' -f1 | wc -c
```

by its result without change of the meaning of the entire program.

2.4.3 UNIX pipe-lining

Most of the information stored on the computers can be interpreted as a stream of bytes, which can be more precisely understood as a bunch of characters representing some encoded information. One of the major concepts of UNIX[28] based systems is based on the idea that "Everything is a file", which allows for storage of the streams in unified way. On the basis of clearly defined data model has been formulated important ideology:

"*This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*"[29]

Quoted philosophy provides simple and common abstraction for operating on the streams of data, which gives better interoperability and transparency. The general advantage of containing common interface for all elements in the system is that programs can operate on data in exactly the same way.

Pipe-lining is a mechanism of connecting output from one program directly into input of an another program. It provides possibility to compose programs into chains, that each of the processing stages is responsible for some transformation of received data. Idea of the pipeline processing has been shown in Listing 2.1.

Concept of pipeline processing has been introduced in order to visualize pure functional data processing. Exemplary chain calculating number of letters in the first part of the sentence has been shown in Listing 2.2.

Thought" supercomputer in one of the Douglas Adams books [27].

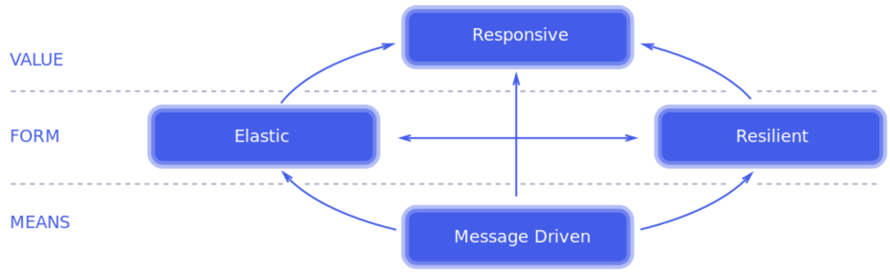


Figure 2.8 Reactive manifesto requirements[32].

2.5 Reactive programming

Reactive Programming[30] (RP) is a programming paradigm designed to simplify process of event-driven applications programming and asynchronous code execution. It is based on the concept of changes propagation and data flows processing, that applications are designed to “react” for incoming events.

2.5.1 Reactive manifesto

In reactive manifesto[31], proposed by community in 2013, have been described four fundamental rules which should fulfil “reactive” application. The main idea was to introduce some clever way in application design, that would rust to increasing demands associated with keywords like scalability, robustness or fault tolerance. Structure of reactive requirements has been shown in Figure 2.8.

In accordance with the diagram, a reactive system can be formulated as:

- **Responsive** - system should always respond for an request in reasonable time.
- **Resilient** - system should stay responsive after occurrence of the failure.
- **Elastic** - system easily scales under heavy load and releases resources when load decreases.
- **Message Driven** - communication in the system is based on asynchronous exchange of messages[9].

[9]Given property was defined in the first version of the manifesto as *Event Driven*.

The formulation of the reactive system concept has been inspired by the largest systems existing in the world. In general manifesto is an aggregation of best practices and principles, which can be used as a hints during design process of the systems. The advisability of this concept can be confirmed by the number of people who signed the manifesto.

2.5.2 Reactive extension

Nowadays concept of RP is mostly associated with reactive frameworks, which are providing reactive programming model to different programming languages. They are combining concepts of iterator and observer patterns together with best practices taken from FP. One of the main types related to RP is called *observable*. For that reason, considerations will start with the explanation of this model.

Let us define single producer and consumer, assuming that consumer is interested in data contained by producer. By way of reminder, iterator pattern allows for traversing through collections and consistent consumption of stored data. Consumer can ask producer for iterator and sequentially *pull* items from it, which will result in visiting all interested elements. During iteration, consumer can obtain information that there is no more data to iterate or catch an error (*thrown* by producer), which will signalize that some problem during iteration has occurred. Given model allows for propagation of three informations:

- Values owned by producer.
- Completion of the processing.
- Error occurrence.

On the other side observer pattern exists. In a similar way it can be used to exchange data between producers and consumers. As it has been described in Section 2.2.2, subject (producer) can notify subscribers (consumers) by execution of registered callbacks.

Although iterator and observer patterns are similar, there is significant difference between them. In concept of **iterator**, **consumer** is controlling notification process by sequentially ***pulling*** data from iterator. In **observer** pattern, **producer** is controlling notification process by ***pushing*** data up to callbacks registered by the

Table 2.1 Differences between iterator pattern and observable.

Event type	Iterator (pull)	Observable (push)
new data	Item next()	onNext(Item)
error occurrence	throws Exception	onError(Exception)
processing completion	not hasNext()	onCompleted()

consumers. The main thing is, that observer pattern does not allow to propagate information concerning the occurrence of an error or a notification completion.

For that reason, reactive extensions are defining additional type called "Observable", which extends existing structure of observer pattern by semantics of iterator pattern. In the new model, instead of registering a single callback for data notification, the consumer can use observable containing 3 separate functions, which can be used to propagate a different kind of information:

- onNext(Item) - is called by producer for every new occurrence of event, that latest value is propagated as function argument.
- onError(Exception) - is called by producer when any error is observed during processing and details concerning thrown exception are forwarded as function argument.
- onCompleted() - is called by producer when there is no more data to notify.

Differences in behaviour between iterator pattern and observable has been shown in Table 2.1.

2.5.3 Asynchronous data stream

Definition of observable from Section 2.5.2 defines syntax for operations, which can be performed between producers and consumers. In order to explain the behaviour of observable, in this section will be introduced a model of *asynchronous data stream*[10], which has been shown in Figure 2.9.

As can be observed, stream is composed from sequentially occurring events (coloured circles), which are modelling next calls of *onNext* function made by producer on

[10]Sometimes also called observable data stream, on the basis of its behaviour.

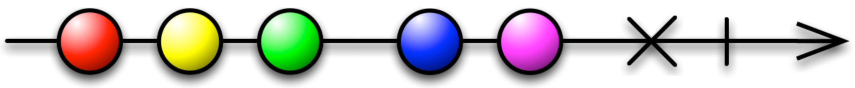


Figure 2.9 Asynchronous data stream.

observable. Each from emitted events is combined with discrete point in time in accordance with definition from Section 2.3.2. Nevertheless its value is not directly exposed to the user[11]. There is no possibility of notifications reordering, because notification model is sequential. The moment of the stream completion has been marked by vertical line, that is realized by *onCompleted* function. It is modelling a moment, when producer finishes notification procedure and will never send events into stream. The final mark (black cross) represents occurrence of an error, which interrupts notification process by analogy to stream completion signal.

Behaviour of stream can be summarized will following formulations:

- Stream can be composed from zero or more events occurring sequentially in time.
- After occurrence either *onCompleted* or *onError* event (not both), next function calls on observable are forbidden.

2.5.4 Reactive operators

On the basis of observable syntax and asynchronous data stream semantics, consumer is able to express its logic in declarative way, by defining how it will *react* for any of incoming events. Created in that way *indisputable contract* allows for usage of reactive operators.

Operators are a functions, which allow to perform operations on asynchronous data streams. They are an universal tools, which can be used for example to create, transform, filter or combine streams. On the basis of assumptions of immutability, operations on streams should be performed without modification of input data. For that reason, most of the operators instead of modifying source streams, are creating new streams with modified content. In Figure 2.10 has been shown *filter* operator. For any event observed in source stream, operator generates event in

[11]Occurrence time can be obtained for example with usage of *timestamp* operator, which generates pairs composed from time point and value of an event.

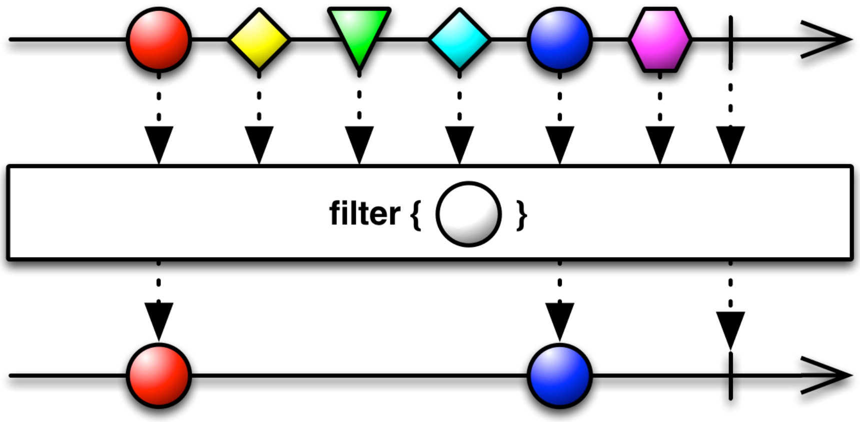


Figure 2.10 Reactive filter operator[33].

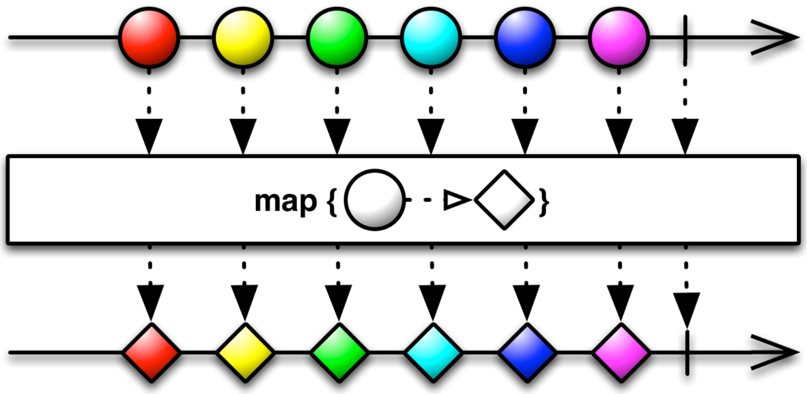


Figure 2.11 Reactive map operator[34].

output stream if and only if event fulfils defined criteria.

Filter operator allows to reduce number of elements in streams, by application of defined predicate. In order to perform modifications of events, transform operators can be used. Usage of most popular *map* operator has been shown in Figure 2.11. As can be seen, shape of each object has been modified from circle to square. Given operator is not breaching assumptions concerning immutability, because as it has been previously mentioned, each value generated in output stream is a copy of the source event with some modification. Map operator allows also for modification of the type of events. That is to say, that newly generated event can express totally different concept in compare to its source.

Up to this point there have been shown only operators operating on single stream

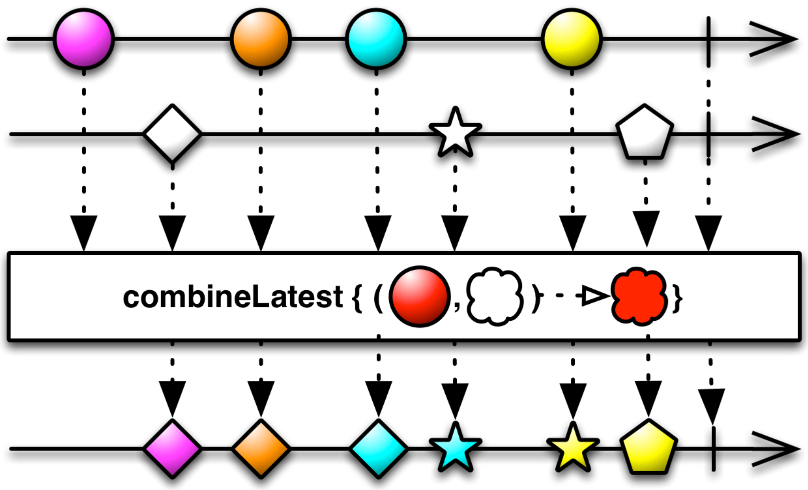


Figure 2.12 Reactive combine latest operator[35].

of data. The huge advantage of reactive paradigm is that operators can be used to operate simultaneously on multiple streams, wherein every operation can be realized in safe and declarative way. In Figure 2.12 has been shown application of *combineLatest* operator, which by analogy to its name is returning latest values observed in any of input streams. As can be observed, order of events occurrence is not important, because synchronization logic is protected by mechanisms defined inside operator. From perspective of the user, relation between streams has been expressed in declarative way.

On the basis of event definition presented in Section 2.3.2, each event is correlated with discrete moment of its occurrence. Owing to given information, reactive operators allow to perform operations based on time. In order to explain some time related functionality, in Figure 2.13 has been shown behaviour of *debounce* operator. As can be seen, operator is monitoring time of incoming events. If in given timespan has not been observed event emittance, then operator is generating last known value from input stream.

According to definition of asynchronous data stream shown in Section 2.5.3, streams can also propagate informations representing emission completion or error occurrence. One of the simplest operators responsible for errors generation has been shown in Figure 2.14(a). Logic realized by operator is similar to *debounce* operator, but instead of generation of last known value, *timeout* operator is signalizing error in stream and finishes notification process[12]. Presented situation can be han-

[12]Interruption of notification process is related to semantics of asynchronous data stream,

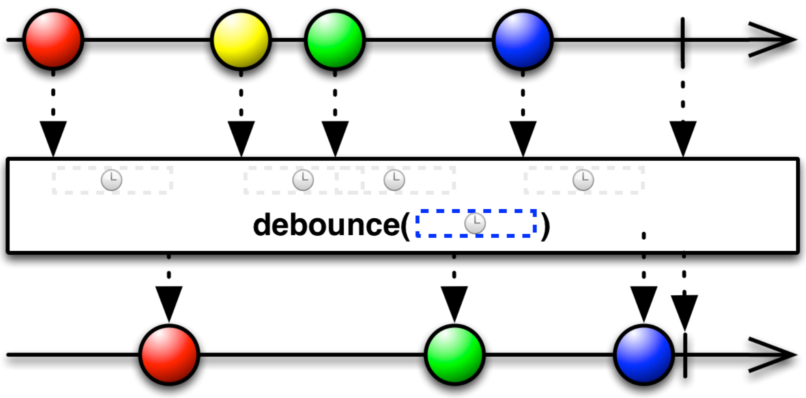


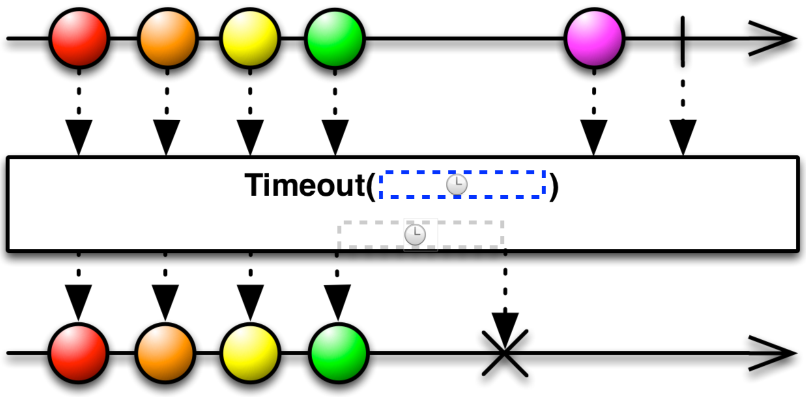
Figure 2.13 Reactive debounce operator[36].

dled with recovery operators, for example with usage of *catch* operator (Figure 2.14(b)). The "graceful shut-down" is performed by "catch" of the *onError* event, emittance of predefined informations and notification completion.

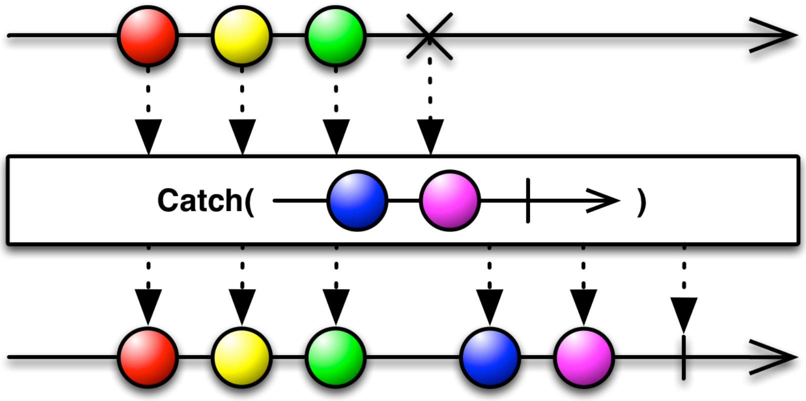
The final property is related to composability of operators. As long operators are processing observables by emitting new ones, they can be composed into chains sequentially realizing defined logic[13]. Chained constructions will be shown in Chapter 5.

which was defined in Section 2.5.3.

[13]Given methodology is similar to pipe-lining concept shown in Section 2.4.3.



(a) Reactive timeout operator[37].



(b) Reactive catch operator[38].

Figure 2.14 Operators related to error generation and recovery operations.

Chapter 3

Distributed software development

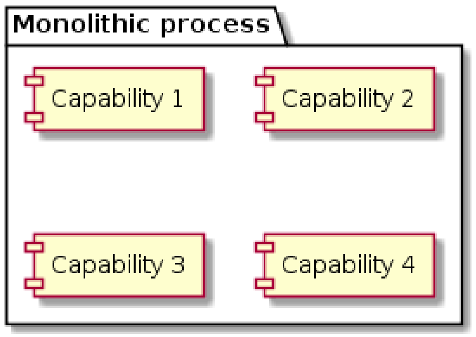
In this chapter will be concerned information related to software development in distributed systems. Considerations will start from formulation of distributed system on the basis of *micro-service architecture*. Afterwards there will be shown one of the most popular abstractions used in process of messages exchange. Finally there will be described middlewares realizing communication in distributed systems.

3.1 Micro-services

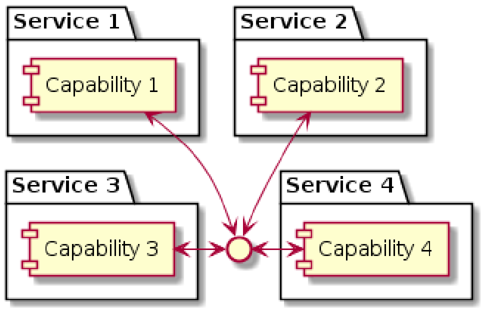
Software development of applications can be realized in various ways from perspective of architecture design. In classic monolithic approach, logic of the application is collected in single executable program, that all developed capabilities are indivisible.

On the other side, the same behaviour can be expressed as group of programs cooperating with each other, that has been visualized in Figure 3.1. The idea of micro-services will be explained with usage of characteristics described in [39] and are most commonly found in industry:

- **Componentization via Services** - components can be replaced and upgraded independently, because logic is separated between services.
- **Organized around Business Capabilities** - development is organized by



(a) Monolithic architecture



(b) Micro-service architecture

Figure 3.1 Comparison of monolithic and micro-service architectures.

small teams and each team is owning its own business area.

- **Products not Projects** - service developers take care of delivered software over full time of production process.
- **Smart endpoints and dumb pipes** - services control process of information exchange, that middleware is providing messages exchange and delivery warranty without control of business process work-flow.
- **Decentralized Governance** - each service can be developed in different technology, which is dedicated for defined kind of problems.
- **Decentralized Data Management** - state of the system is distributed over services. Each individual service does not share directly stored informations with external world.
- **Infrastructure Automation** - process of services delivery is automated. Services are tested separately with *System Component Tests* (SCT) and together with other components during *System Integration Tests* (SIT).
- **Design for failure** - system is able to perform recovery operations, when some of the services fail.[1]
- **Evolutionary Design** - system can be easily extensible with new features. Software modifications will be performed without reorganization of entire system structure.

[1]Interesting method of system verification has been presented by Netflix company. There has been used a tool called *Chaos monkey*[40], which randomly disables services and verifies if system is resilient.

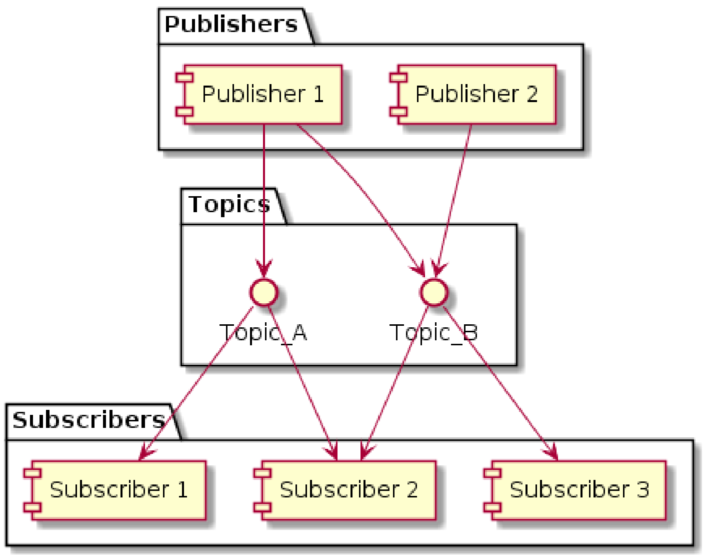


Figure 3.2 Visualization of publish-subscribe paradigm.

Distributed style of software development is characteristic for concepts like CBD or *micro-service architecture*. In case of micro-services, designed model can be interpreted as specialisation for *Service Oriented Architecture*[41] (SOA), wherein each service logic is expressed in accordance with *Single Responsibility Principle*[42] (SRP). Micro-services as a whole require additional mechanisms of information exchange, which will be discussed in more details in next sections.

3.2 Publish-subscribe pattern

In order to simplify process of information exchange in distributed systems there have been created message patterns, which are providing useful abstractions over topology of the network. One of the most popular abstractions is called *Publish-subscribe pattern*[43] (Pub/Sub). In Figure 3.2 has been shown structure of the pattern, representing relations between participants in the network.

Proposed pattern is based on abstraction, which provides communication interface between senders (publishers) and receivers (subscribers). The exchange of information (messages) is realized with usage of strongly typed communication channels (topics). Subscriber who is interested in some type of information performs a subscription in given topic and starts to receive information from publishers. From perspective of subscriber, information about sender of the messages is not important. Communication is realized without knowledge about topology of the network and existence of other components, which allows for couple loosing between publishers and subscribers and simple reorganization of the network structure[2].

3.3 Data distribution service

According to the concept of the micro-service architecture described in Section 3.1, there is a need for solutions realizing communication in distributed systems. One of the standardized protocols, realized in Pub/Sub and broker-less design, is called *Data Distribution Service*[44] (DDS). The aim of the protocol is to provide real-time, high-performance and reliable communication method in critical time systems like air-traffic control, military or medical devices. DDS is a suggested middleware for robotic applications[45].

The protocol is composed of two independently specified standards. First of them is responsible for providing abstraction in users space[44], second as real-time transport layer[46]. DDS as end-to-end solution provides an *Interface Description Language*[47] (IDL), which allows to uniformly represent data types used in inner-process communication. The main advantage of this standard is *Quality of Service* (QOS), which is a set of configurable parameters useful in control of DDS behavior[48].

Most important from the perspective of further considerations is an explanation of transport layer of the protocol, which is called *Real Time Publish Subscribe*[46] (RTPS). According to the information provided in specification[46], communication between publishers and subscribers is realized by default with usage of *User Datagram Protocol*[49] (UDP). It allows for highly efficient and not reliable exchange of messages between components. It also supports simultaneous delivery of single datagram to multiple receivers (multi-casting). It is important to notice, that reliable communication has been realizing not reliable transport layer. Given property has been achieved by application of additional messages, in which each sent datagram must be acknowledged by its receiver. In order to exchange information about the existence of participants in the network, each vendor is responsible for the implementation of *Participant Discovery Protocol*[46] (PDP) and *Endpoint Discovery Protocol*[46] (EDP).

[2]Concept is similar to to observer pattern described in Section 2.2.2 but is realized on inner-process communication level.

With reference to Pub/Sub concept, communication is realized with the usage of *DataWriters* and *DataReaders*. Each of them allows for the fully asynchronous exchange of information in *Peer-to-peer*[50] (P2P) architecture, that makes this protocol a perfect source of asynchronous data streams[3].

3.4 Robot Operating System

Robot Operating System[51] (ROS) is a programming platform designed to simplify the process of robots development. It is composed of wide range set of tools and libraries, that can be used to prevent developers from programming its own software from scratch.

On the operating system level, ROS provides inner-process communication mechanisms, packages management and hardware abstractions. Algorithms, that solve most common robotic problems, are shared on the application level. Framework also contains additional tools, that are useful in debugging or system state visualisation.

Software development in ROS framework is based on separated applications called *nodes*[52], that realize CBD concept. In communication between nodes can be used Pub/Sub, in which routing of messages is performed by ROS message broker called *roscore*. An important attribute is that each component can be developed in different programming language[4], because nodes communicate with usage of language independent protocol. Structure of exemplary ROS system graph has been shown in Figure 3.3.

In the area of each node can be created publishers and subscribers, that can be used to exchange messages between other components. There is also possibility to create *Remote Procedure Call*[53] (RPC) services, which can be used to expose internal functionalities of the nodes. Given approaches allow to apply advantages of distributed service architecture described in Section 3.1.

In the second version of ROS, community decided to switch custom implementation of transport layer and interface representation into end-to-end solution[54]. For that reason, DDS has been chosen as default middleware, which by its properties can be substituted in place of existing solution.

[3]Described in Section 2.5.3.

[4]Right now ROS is officially supporting *C++*, *Python* and *Lisp* languages.

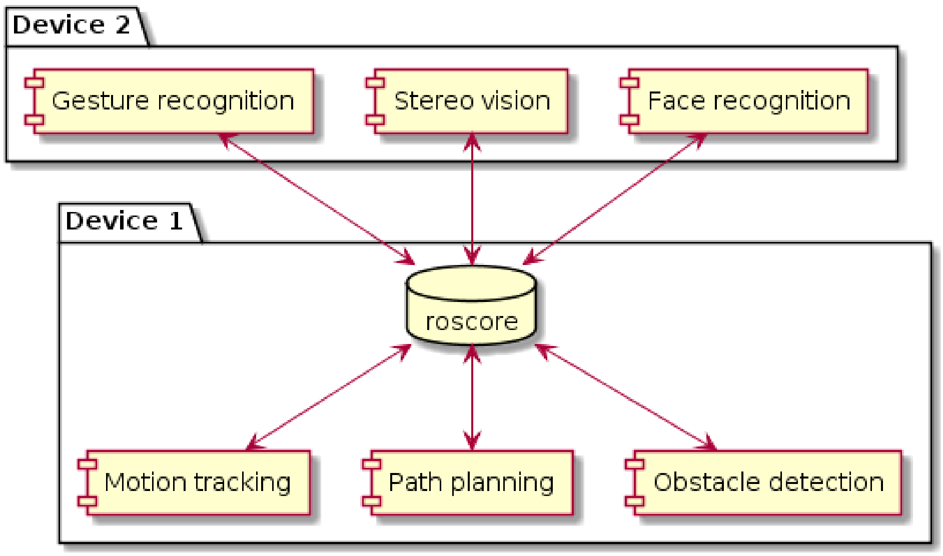


Figure 3.3 Visualization of exemplary ROS system structure.

Chapter 4

Library implementation

In the following chapter will be introduced architecture and implementation of developed library, which combines RP and FP programming paradigms with RTPS protocol.

4.1 Foreword and assumptions

There have been performed several assumptions during implementation of the library. Described in Section 3.3 RTPS protocol has been chosen to model asynchronous data streams. As a indirect vendor of RTPS has been chosen ROS (second version), because it provides unified programming interface and allows for re-usability of developed library in existing robotic applications.

Furthermore has been used C++ implementation of *ReactiveX* [55][56] (Rx) library, in order to use functional and reactive paradigms. On the basis of information provided by the authors of the Rx library, framework does not strictly realize the concept of FRP caused by the relaxation of continuous time assumption. Regardless of given divergence, Rx can be used to model reactive and functional concepts in similar way to more restricted interpretations like in [57]. Although Rx framework does not model continuous time behaviours, RTPS protocol can be used as a source of asynchronous data streams, because considerations are focused only on processing of discrete events obtained from the protocol.

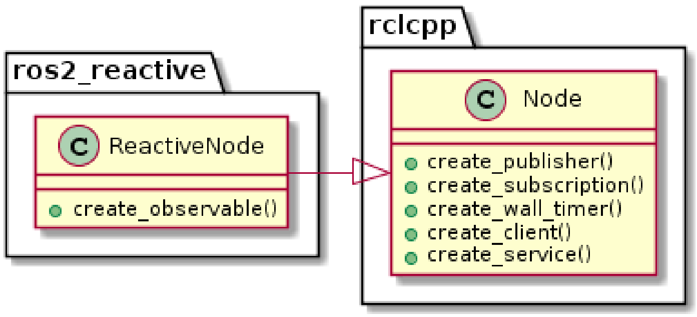


Figure 4.1 Class diagram for created reactive extension in UML notation.

4.2 Architecture

As a base type has been chosen *rclcpp::Node*[1], which among other things allows to create subscriptions, publishers or services. In newly created *ReactiveNode*, set of available methods has been extended by new functionality, which allows to create reactive observables compatible with reactive framework. Relations between those types have been shown in Figure 4.1.

On the basis of the inheritance property, *ReactiveNode* behaves exactly in the same way as its base type. As long as interface of base class is available in the scope of the derived class, proposed type can be integrated into existing structure of ROS applications without code modifications. Interface of *create_ observable* function is similar to interface of *create_ subscription* method. The difference is related to number of input arguments, because subscription callback has been removed from *create_ observable* function parameters.

Snippets explaining usage of developed library have been shown in Section 4.3.

4.3 Interface usage

Library can be used in similar way as standard ROS node. Method of ROS node creation has been shown in Listing 4.1. As can be observed, created object allows to create subscriptions parametrized with callback functions. Provided by the user

[1]Concept of ROS node has been described in Section 3.4.

handler will be called for every incoming message. As a return value user obtains subscription handler, which keeps registered callback alive as long as handler is not destroyed[2].

Listing 4.1 ROS node creation.

```
auto node = std::make_shared<rclcpp::Node>("Node_name");
```

Listing 4.2 ROS subscription creation.

```
auto sub = node->create_subscription<std_msgs::msg::String>(
    "topic",
    [](std_msgs::msg::String::UniquePtr msg)
    {
      std::cout << "Data:␣" << msg->data << std::endl;
    }
);
```

The same behaviour can be obtained by substitution of developed *ReactiveNode*, which implements all functions of standard ROS node (Listing 4.3). In particular case, there is created reactive observable of type *String* from defined topic. As a return value, user obtains *Observable* of defined type (String), representing each received message as new discrete event.

Listing 4.3 Reactive node creation.

```
std::shared_ptr<ros2_reactive::ReactiveNode> node =
    std::make_shared<ros2_reactive::ReactiveNode>("Node_name");
```

Listing 4.4 Reactive observable creation.

```
rxcpp::observable<std_msgs::msg::String> observable =
    node->create_observable<std_msgs::msg::String>("topic");
```

Interfaces of *create_ subscription* (Listing 4.2) and *create_ observable* (Listing 4.4) functions differ in number of input arguments. In RP with usage of Rx framework, user can provide three independent callbacks handling different types of observable function calls[3]. Exemplary subscription with usage of multiple callbacks has been shown in Listing 4.5.

Created observable will be deactivated until first call of subscribe method, which means that library will not perform any network operations as long as there are

[2]With reference to *Resource Acquisition Is Initialization*[58] (RAII) programming idiom.
[3]Methods available for *Observable* type have been described in Section 2.5.2.

Listing 4.5 Reactive subscription with usage of multiple callbacks.

```
rxcpp::observable<std_msgs::msg::String> observable =
    node->create_observable<std_msgs::msg::String>("topic");

observable.subscribe(
    [](std_msgs::msg::String const& msg)
    {
      std::cout << "Data:␣" << msg.data << std::endl;
    },
    [](std::exception_ptr error)
    {
      std::cout << "On␣error:␣" << error << std::endl;
    },
    []()
    {
      std::cout << "On␣completed" << std::endl;
    });
}
```

not registered observers[4]. More advanced examples of the library usage have been shown in Chapter 5.

[4]More informations about programming with usage of Rx library can be found on project web-page[55]

Chapter 5

Reactive robotic application

In this chapter will be proposed exemplary application constructed on the basis of library explained in Chapter 4. Application will be responsible for realisation of simple robotic task, with the main focus on methods of data processing with reference to reactive methodology. In order to simplify the concept of informations processing, it has been assumed, that entire data received by application will be derived from virtual sources.

5.1 Problem definition

Lets consider two-link manipulator shown in Figure 5.1. Structure of the robot is composed from two link lengths (l_1 and l_2) and two angles (θ_1 and θ_2). Position of the effector can be expressed by following equations:

$$\begin{cases} x = l_1 * cos(\theta_1) + l_2 * cos(\theta_1 + \theta_2) \\ y = l_1 * sin(\theta_1) + l_2 * sin(\theta_1 + \theta_2) \end{cases} \tag{5.1}$$

Proposed application will realize simple monitoring task by a cooperation with presented manipulator model. System will be responsible for collecting configuration parameters of manipulator and performing some computations for robot operator. Logic of the application has been defined with usage of *User stories*[59], which are part of Agile development methodologies[60].

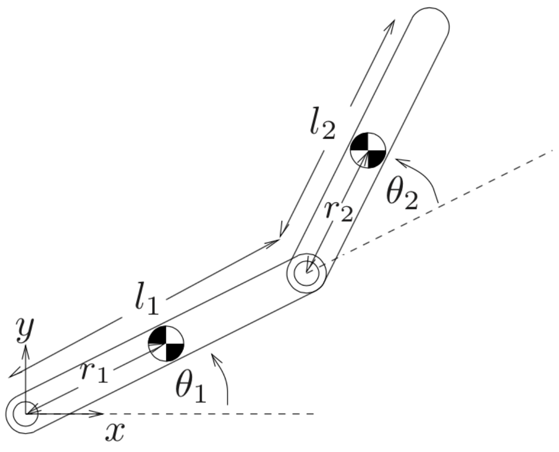


Figure 5.1 Structure of two-link manipulator[61].

1. As a robot operator, I am able to observe most recent speed and position of robots effector.
   - Position of a robot effector is a pair containing x and y coordinates of effector.
   - Speed of a robot effector is a constant velocity measured by distance between last two observed points divided by time of the movement.
     - Distance between last two observed points is an euclidean metric obtained of two robot effector positions.
2. As a robot operator, I am able to turn off engines when operator becomes incapacitated.
   - Robot operator becomes incapacitated when he release *Dead Man's Switch* 3 times for time longer than 3 seconds.
   - Turn off the engines means that application will send message to engine controller with command "turn_off".

5.2 Architecture of application

In this section will be shown architectural of the application, which was inspired by presented stories implementation. Logic of events processing has been visualized

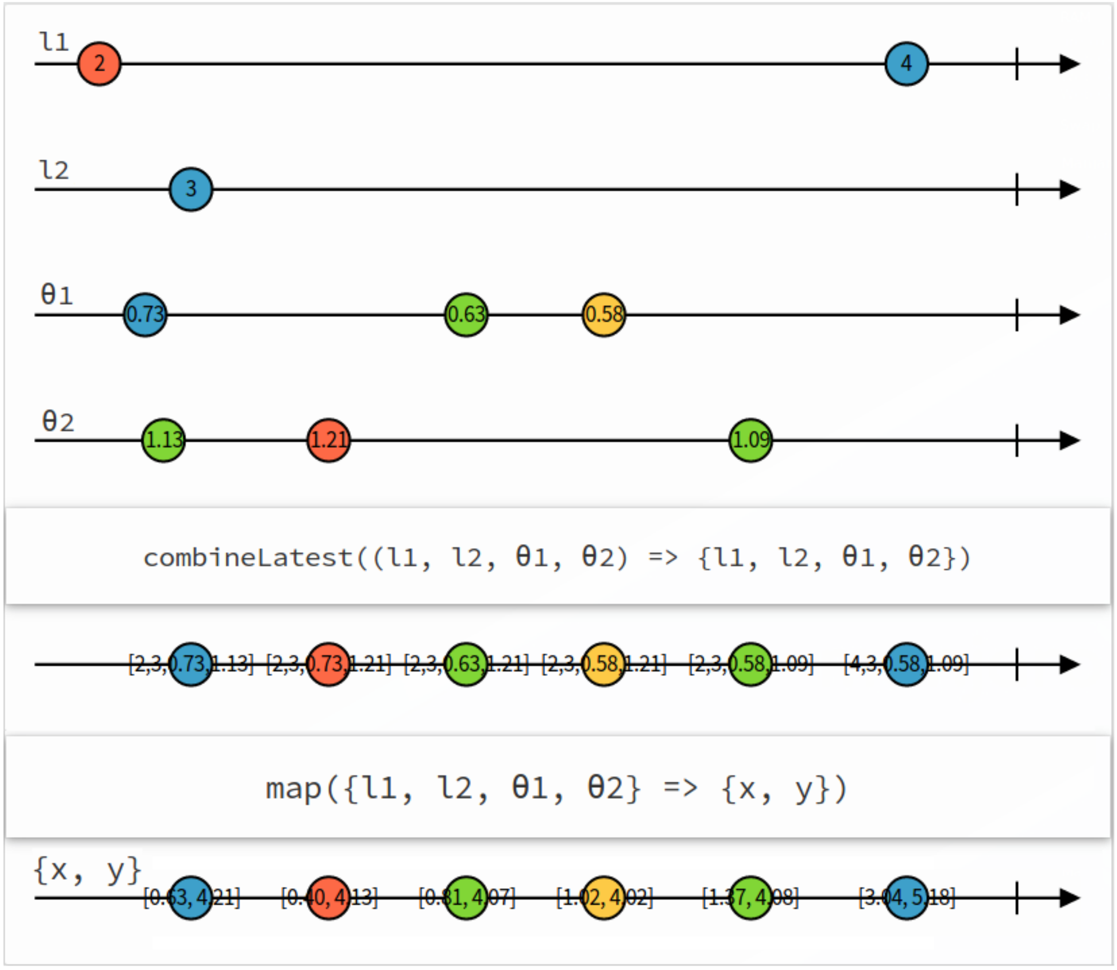


Figure 5.2 Reactive manipulator configuration processing into effector position.

with usage of marble diagrams[62] without exposing implementation details.

5.2.1 Effector coordinates

With reference to Equation 5.1, computation of effectors position requires 4 input parameters (l_1,l_2, θ_1 and θ_2). On the basis of assumptions described in Section 4.1, each change of parameter from manipulator configuration will be obtained from virtual source. In particular case, it will be modelled by four independent ROS topics, delivering latest values for each from given manipulator parameters. Streams have been visualized in Figure 5.2.

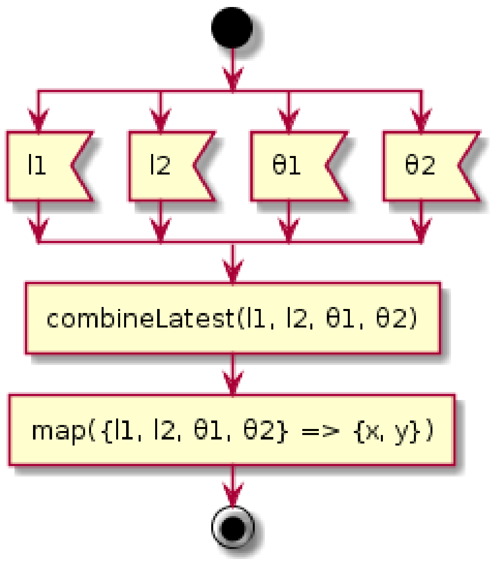

l1
l2
θ1
θ2
combineLatest(l1, l2, θ1, θ2)
map({l1, l2, θ1, θ2} => {x, y})

Figure 5.3 Recipe for effector position expressed by manipulator configuration parameters.

It is important to notice, that each value can change independently at any discrete point in time. In order to always have latest position of effector, any change observed in input streams should trigger coordinates recalculation. For that reason, streams have been combined with usage of *combineLatest* operator. As can be observed in Figure 5.2, for each observed event, in any of four input streams, *combineLatest* operator generates new event, which contains latest values aggregated from all input streams. Given declarative definition models stream of most recent manipulator parameters[1].

Just created stream can be used to calculate effector coordinates. For that reason has been used *map* operator, which transforms all events emitted by *combineLatest* operator ($\{l_1, l_2, \theta_1, \theta 2\}$) into pair $\{x, y\}$, containing x and y position of effector. Transformation is performed with usage of Equations 5.1.

As can be noticed, logic of given story has been expressed in declarative way, by clever recipe explaining how effector coordinates are expressed by manipulator configuration. In order to formalize transformations, logic has been visualized as activity diagram in Figure 5.3. Just modelled stream of effector positions will be reused in further user stories.

[1]Additional change of first link length l_1 has been shown in order to explain behaviour of operator.

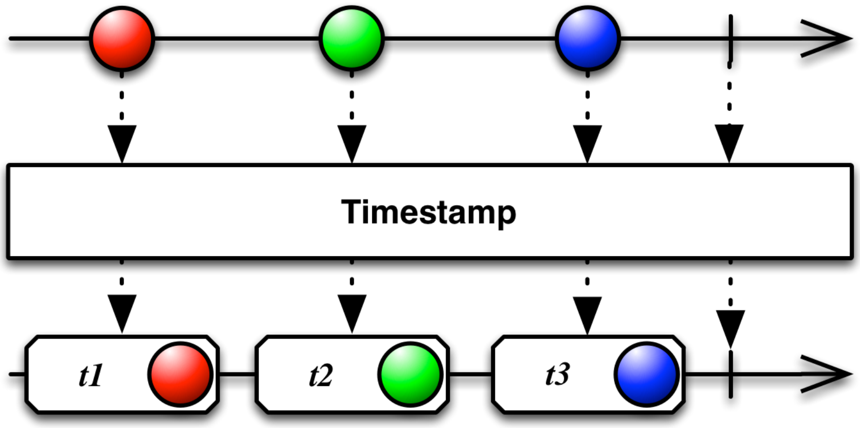


Figure 5.4 Reactive timestamp operator[63].

5.2.2 Velocity of effector

Second story will be realized on the basis of data obtained from calculation of effectors position. In accordance to the problem formulation, velocity of the effector will be represented as distance between last two known positions and divided by movement time. As it has been mentioned, distance will be calculated as an euclidean norm shown in Equation 5.2.

$$d(p,q) = d(q,p) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{5.2}$$

Before calculation of the distance, to each observed position event will be attached time-stamp of its emittance, which will be realized by application of *timestamp* operator shown in Figure 5.4. Formulated in that way stream will be transferred as an input to *pairwise* operator (Figure 5.5), which will merge last two values observed in the stream. Finally created pairs will be recalculated into velocity by calculation of the distance between points and division by the difference between events emittance[2]. In order to not generate unnecessarily values in situation when calculated velocity not changed, will be applied *distinct_until_changed* operator shown in Figure 5.6. Final flow of the stream has been visualized in Figure 5.7.

[2]Given operation is similar to calculation of position with map operator described in Section 5.2.1. For that reason it will be not visualized again.

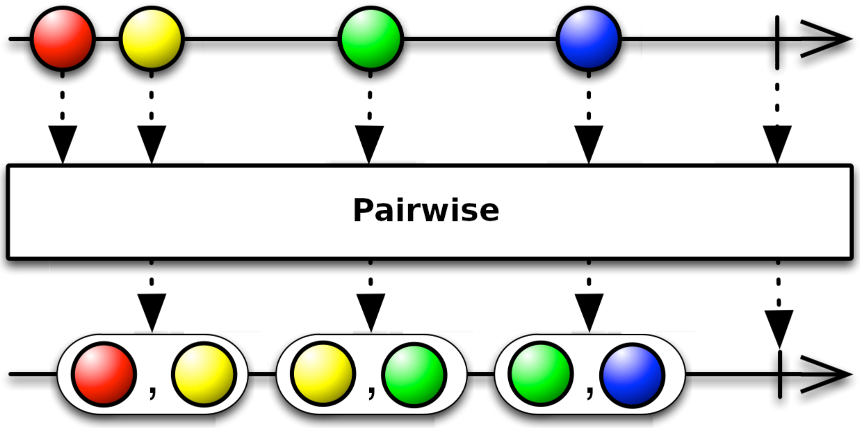


Figure 5.5 Reactive pairwise operator.

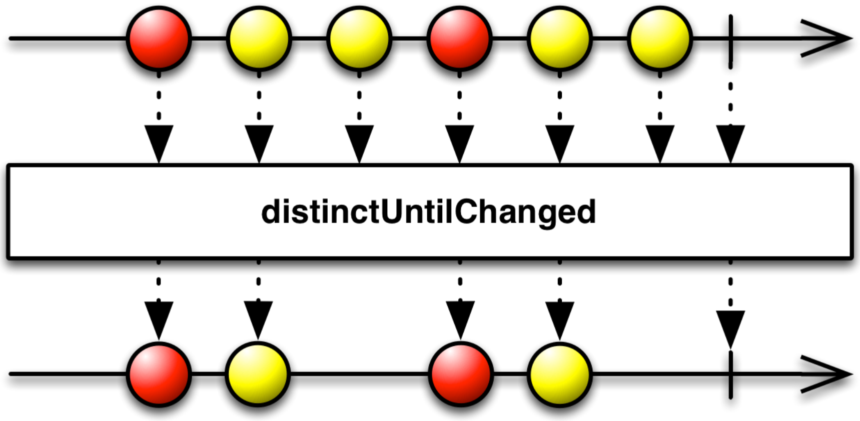


Figure 5.6 Reactive distinct until changed operator[64].

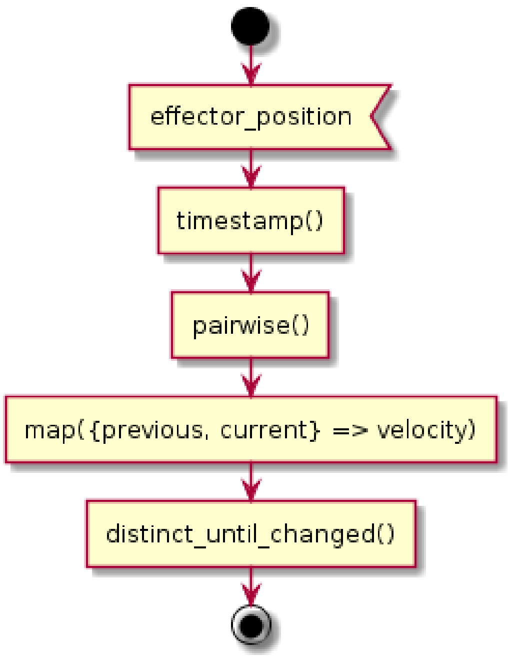


Figure 5.7 Recipe for velocity expressed by effector position events.

5.2.3 Operator activity monitoring

Last functionality is related to turning off engines, when operator becomes incapacitated. Presence of the operator will be simulated by virtual *Dead Man's switch* (Figure 5.8), which will send with high frequency messages representing press of the button.

Simulation of continuously pressed button will result in thousands of messages per second, which can significantly reduce performance of the entire application. For that reason at the beginning will be applied *throttle_ last* operator (Figure 5.9), which will emit latest value observed in the stream with 500 milliseconds interval.

Next part of processing will concern activity of the operator, where 3 seconds interval will be guarded by shown in Figure 2.14(a) *timeout* operator. In situation when will not be observed event in defined interval, then *timeout* operator will signalize error occurrence and finish notification process. In accordance with semantics of *observable* introduced in Section 2.5.3, in stream can not be emitted events after occurrence of an error. In order to perform recovery operation will be applied *retry* operator (Figure 5.10), which will "catch" *onError* function call and

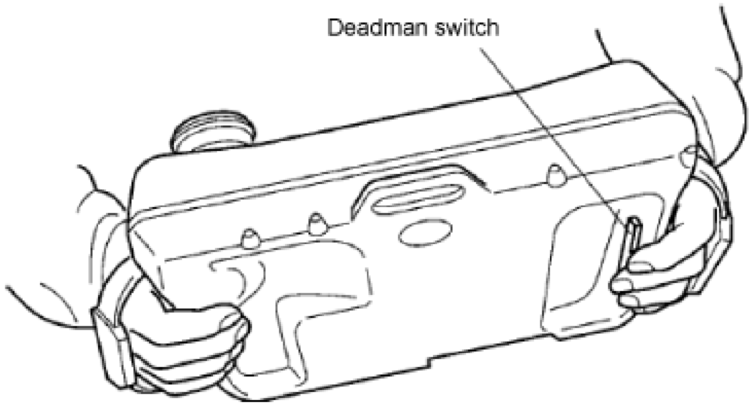


Figure 5.8 Dead Man's switch visualisation[65].

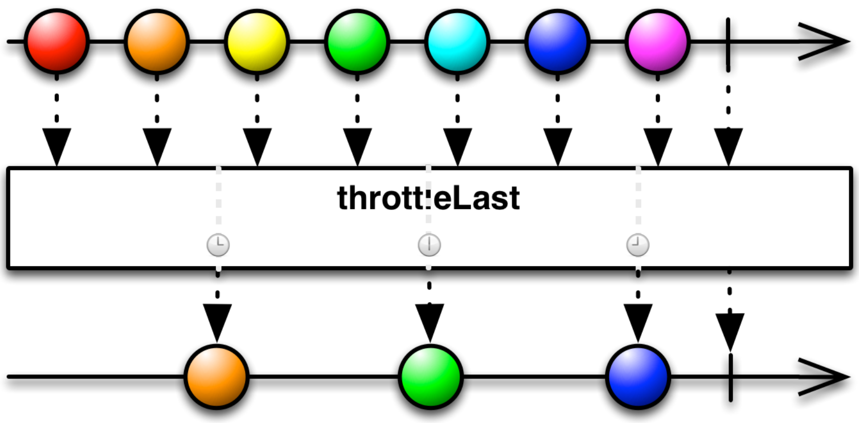


Figure 5.9 Reactive throttle last operator[66].

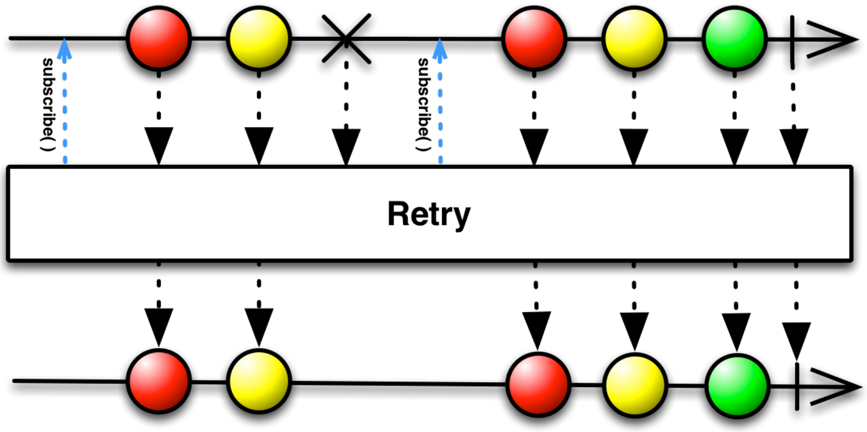


Figure 5.10 Reactive retry operator [67].

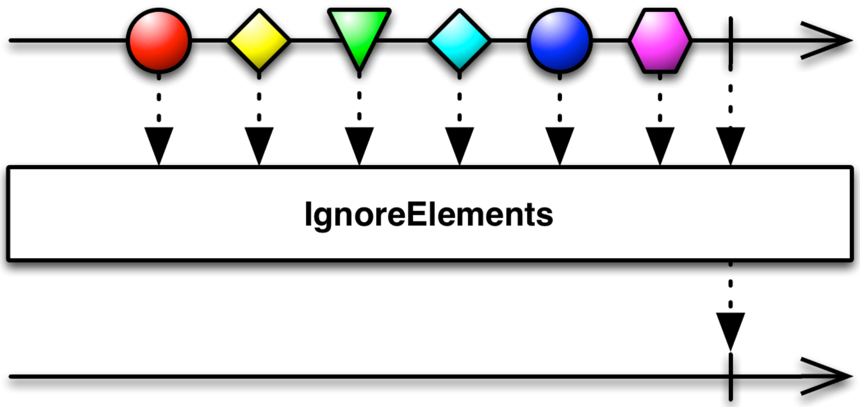


Figure 5.11 Reactive ignore elements operator[68].

perform resubscribe operation. *Repeat* operator will execute subscription procedure 3 times. If in the stream will be observed another error, then operator will propagate this information to the subscribers.

As can be observed, since moment of application of the *timeout* operator, processing logic is not related to events emitted in the streams. For that reason as a second performance optimization will be applied *ignore_ elements* operator (Figure 5.11), which will remove all events from the stream and will propagate only informations about error occurrence and stream completion. Recipe representing operator activity has been shown in Figure 5.12.

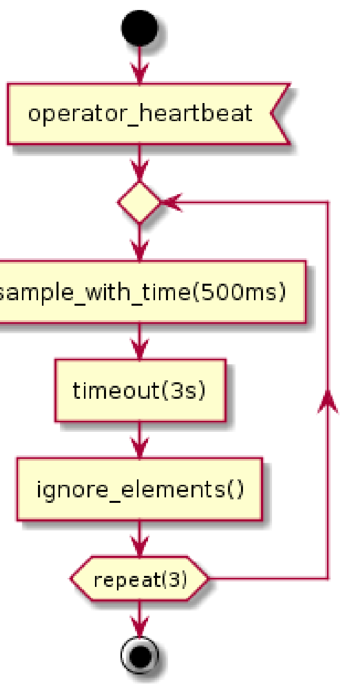


Figure 5.12 Recipe for operator activity expressed by activity signal.

5.3 Implementation details

In this section will be presented implementation details of the described user stories. Structure of the application has been shown in Listing 5.1. It has been assumed, that for each implemented story is using common *ReactiveNode* instance as a source of *observables*. Lifetime of the node is managed by internal ROS mechanisms.

5.3.1 Effector coordinates

Firstly have been declared sources of the necessary application inputs shown in Listing 5.2. Each of them represents independent and asynchronous ROS topic of defined *Float64* message type.

Secondly have been defined observable of effector positions shown in Listing 5.3. With reference to the architecture specified in Section 5.2.1, latest state of the

Listing 5.1 Common representation of the reactive node and application structure.

```cpp
#include <std_msgs/msg/float64.hpp>
#include <std_msgs/msg/string.hpp>
#include <std_msgs/msg/bool.hpp>

#include <rclcpp/rclcpp.hpp>
#include <ros2-reactive/ReactiveNode.hpp>

#include <chrono>
#include <iostream>

int main(int argc, char* argv[])
{
  rclcpp::init(argc, argv);

  auto node = std::make_shared<ros2_reactive::ReactiveNode>("RxNode");

  /* application code */

  rclcpp::spin(node);
  rclcpp::shutdown();
}
```

Listing 5.2 Observables representing configuration of manipulator.

```cpp
rxcpp::observable<std_msgs::msg::Float64> l1 =
    node->create_observable<std_msgs::msg::Float64>("l1");
rxcpp::observable<std_msgs::msg::Float64> l2 =
    node->create_observable<std_msgs::msg::Float64>("l2");
rxcpp::observable<std_msgs::msg::Float64> theta1 =
    node->create_observable<std_msgs::msg::Float64>("theta1");
rxcpp::observable<std_msgs::msg::Float64> theta2 =
    node->create_observable<std_msgs::msg::Float64>("theta2");
```

Listing 5.3 Declarative specification of effector position.

```
rxcpp::observable<std::tuple<double, double>> x_y = l1
  .combine_latest(l2, theta1, theta2)
  .map([](std::tuple<std_msgs::msg::Float64,
                     std_msgs::msg::Float64,
                     std_msgs::msg::Float64,
                     std_msgs::msg::Float64> const& latest_values)
  {
    std_msgs::msg::Float64 l1, l2, theta1, theta2;
    std::tie(l1, l2, theta1, theta2) = latest_values;

    std::cout <<  "### Received new data: {"
              << l1.data << ", "
              << l2.data << ", "
              << theta1.data << ", "
              << theta2.data << "}" << std::endl;

    double x = l1.data*std::cos(theta1.data) +
               l2.data*std::cos(theta1.data + theta2.data);
    double y = l1.data*std::sin(theta1.data) +
               l2.data*std::sin(theta1.data + theta2.data);
    return std::make_tuple(x, y);
  })
```

manipulator is expressed by composition of four input streams with usage of *combine_latest* operator. After that, defined stream is forwarded to *map* operator, in which parameters are recalculated into pair of x and y coordinates. Just created recipe is saved into x_y observable of latest position of the effector.

Final part of the user story is related to user notification process. For that reason, single subscriber interested in coordinates has been defined, which has been shown in Listing 5.4. As can be seen, defined subscriber is providing single callback function responsible for displaying of the values. There are not registered additional callback function, because there is assumption, that created stream is infinite and faultless.

5.3.2 Velocity of effector

Second story has been implemented on the basis of observable defined in Section 5.3.1. As can be seen in Listing 5.5, at the beginning observable representing latest effector coordinates is reused. On the basis of this source of data, stream is proceed

Listing 5.4 Definition of subscriber interested in effector position.

```
x_y.subscribe(
  [](std::tuple<double, double> const& tuple)
  {
    double x, y;
    std::tie(x, y) = tuple;

    std::cout << "Position x: " << x << " and y: " << y << std::endl;
  });
```

by *timestamp* and *pairwise* operators. After processing, type of the events is modified and contains pairs representing previous and current states of the stream. Each sub-pair is composed from two effector coordinates and a *time_point*, which represents state in the moment of coordinates calculation. Next, last two known states are recalculated into velocity. As a final operation is used *distinct_until_changed* operator, which removes duplicates from the stream. Just created recipe is saved into *speed* observable and is used as source of the data for subscription shown in Listing 5.6. Subscription using three separated callbacks is used to handle informations concerning error occurrence and notification completion.

5.3.3 Operator activity monitoring

Final functionality concerning monitoring of operator activity has been shown in Listing 5.7. As can be observed, logic of the processing is realized directly from output of *create_observable* function call. It is possible, because created observable is not used in next parts of the code and there is no need to store it in separated variable. Logic is implemented by application of *sample_with_time*[3] and *timeout* operators. As can be observed, each operator is executed on dedicated thread, which allows for parallel and non-blocking execution of internal timers. In order to visualize different stages of processing, in critical places is used *tap* operator, which for each received signal calls appropriate function and passes signal on.

Operator activity observable activates by subscription call shown in Listing 5.8. As can be observed, subscriber is composed only from two registered callbacks, because created stream can not finish successfully. For first registered function can be observed similar situation, because as a consequence of *ignore_elements*

[3]In C++ implementation of Rx framework *throttle-last* operator is realized by *sample_with_time* function.

Listing 5.5 Definition of effector velocity.

```cpp
using time_point =
    rxcpp::schedulers::scheduler::clock_type::time_point;

rxcpp::observable<double> speed = x_y
  .timestamp()
  .pairwise()
  .map([](std::pair<std::pair<std::tuple<double, double>, time_point>>
          const& previous_and_current_positions_with_time)
  {
    std::pair<std::tuple<double, double>, time_point> previous;
    std::pair<std::tuple<double, double>, time_point> current;
    std::tie(previous, current) =
        previous_and_current_positions_with_time;

    double x_diff =
        std::get<0>(current.first) - std::get<0>(previous.first);
    double y_diff =
        std::get<1>(current.first) - std::get<1>(previous.first);

    double distance =
        std::sqrt(std::pow(x_diff, 2) + std::pow(y_diff, 2));
    double time =
        std::chrono::duration_cast<std::chrono::milliseconds>(
            current.second - previous.second).count() / 1000.0;

    return distance/time;
  })
  .distinct_until_changed();
```

Listing 5.6 Definition of velocity subscriber.

```cpp
speed.subscribe(
  [](double const& speed)
  {
    std::cout << "Speed:␣" << speed << "␣[m/s]" <<std::endl;
  },
  [](std::exception_ptr)
  {
    std::cout << "Speed␣calculation␣error␣!!!" << std::endl;
  },
  []()
  {
    std::cout << "Speed␣calculation␣completed" << std::endl;
  }
);
```

Listing 5.7 Definition of operator activity stream.

```
node->create_observable<std_msgs::msg::Bool>("operator_heartbeat")
   .tap(
    [](std_msgs::msg::Bool const&)
    {
      std::cout << "###␣Received␣signal␣from␣operator" << std::endl;
    })
  .sample_with_time(rxcpp::synchronize_new_thread(),
                    std::chrono::milliseconds(500))
  .timeout(rxcpp::synchronize_new_thread(), std::chrono::seconds(3))
  .tap(
    [](std_msgs::msg::Bool const&)
    {
      std::cout << "Operator␣is␣alive" << std::endl;
    },
    [](std::exception_ptr)
    {
      std::cout << "Not␣received␣signal␣from␣operator..." << std::endl;
    })
  .ignore_elements()
  .retry(3)
```

operator application, stream is not propagating events at this level. Turn off engines procedure is realized by send of dedicated ROS message with usage of standard ROS publisher.

5.4 Execution in testing environment

Correctness of described implementation has been verified in testing environment. Standard output of the application has been shown in Listing 5.9. Time point of information generation (milliseconds) has been marked with square brackets. Information concerning occurrence of external event has been marked with ### symbol.

On the basis of presented output, following conclusions can be drown:

- Application is waiting for collection of at least one value for each manipulator parameter, which is a characteristic behaviour for *combine_latest* operator.
- Each occurrence of event representing new manipulator parameter triggers

Listing 5.8 Subscription for operator activity stream.

```
.subscribe(
[](std_msgs::msg::Bool const&)
{
  std::cout << "It␣will␣be␣never␣called␣!!!" << std::endl;
},
[&node](std::exception_ptr)
{
  std::cout << "Operator␣is␣sleeping␣!!!" << std::endl;
  std::cout << "Turning␣off␣engines..." << std::endl;

  auto engines =
      node->create_publisher<std_msgs::msg::String>("engines");
  std_msgs::msg::String emergency_message;
  emergency_message.data = "turn_off";

  engines->publish(emergency_message);
});
```

recalculation of the position and speed of effector.

- Omission of the speed values induced by *distinct_until_changed* operator is visible in frames 4243 and 4444.
- Behaviour of suppressing *sample_with_time* operator has been confirmed in frames 2010 and 2510, in which time difference between events is equal to specified 500 milliseconds interval.
- Suspected observation of first operator signal disappearance has been observed by *timeout* operator in frame 7510, exactly 3 seconds after last emittance visible in frame 4510. After three unsuccessful re-subscriptions from frames 10512 and 13513, signal concerning emergency stop has been sent to engines controller, that is a consequence of application of *retry* operator and registered subscription callback.

5.5 Final word

Quoted examples have been chosen in order to show advantages of reactive approach. In Sections 5.2.1 and 5.3.1 has been shown composability of streams, which has been realized by application of combining operators. Time operations

Listing 5.9 Output from exemplary execution of application.

```
[1941] ### Received new data: {2, 3, 0.785398, 0.785398}
[1942] Position x: 1.41421 and y: 4.41421
[1942] ### Received signal from operator
[1991] ### Received signal from operator
[2010] Operator is alive
[2041] ### Received signal from operator
[2091] ### Received signal from operator
[2141] ### Received signal from operator
[2192] ### Received signal from operator
[2242] ### Received signal from operator
[2292] ### Received signal from operator
[2342] ### Received signal from operator
[2392] ### Received signal from operator
[2510] Operator is alive
[3443] ### Received new data: {2, 3, 1.91986, 0.785398}
[3443] Position x: −3.40296 and y: 3.14725
[3443] Speed: 3.31845 [m/s]
[3943] ### Received new data: {2, 3, 1.91986, 1.7854}
[3943] Position x: −3.21994 and y: 0.276522
[3943] Speed: 5.75311 [m/s]
[4143] ### Received new data: {2, 3, 1.91986, 1.7854}
[4143] Position x: −3.21994 and y: 0.276522
[4143] Speed: 0 [m/s]
[4243] ### Received new data: {2, 3, 1.91986, 1.7854}
[4243] Position x: −3.21994 and y: 0.276522
[4444] ### Received new data: {2, 3, 1.5708, 1.7854}
[4444] Position x: −2.93119 and y: 1.36111
[4444] Speed: 5.61186 [m/s]
[4444] ### Received signal from operator
[4510] Operator is alive
[6444] ### Received new data: {2, 3, 2.35619, 1.7854}
[6444] Position x: −3.03513 and y: −1.11019
[6444] Speed: 1.23674 [m/s]
[7510] Not received signal from operator...
[10512] Not received signal from operator...
[13513] Not received signal from operator...
[13513] Operator is sleeping !!!
[13513] Turning off engines...
```

and re-usability of observables have been shown in Sections 5.2.2 and 5.3.2, in which operations have been performed in the area of single stream of data. Final example presented in Sections 5.2.3 and 5.3.3 showed error propagation through streams and methods of error handling. Summarized architecture of the application has been shown in Figure 5.13.

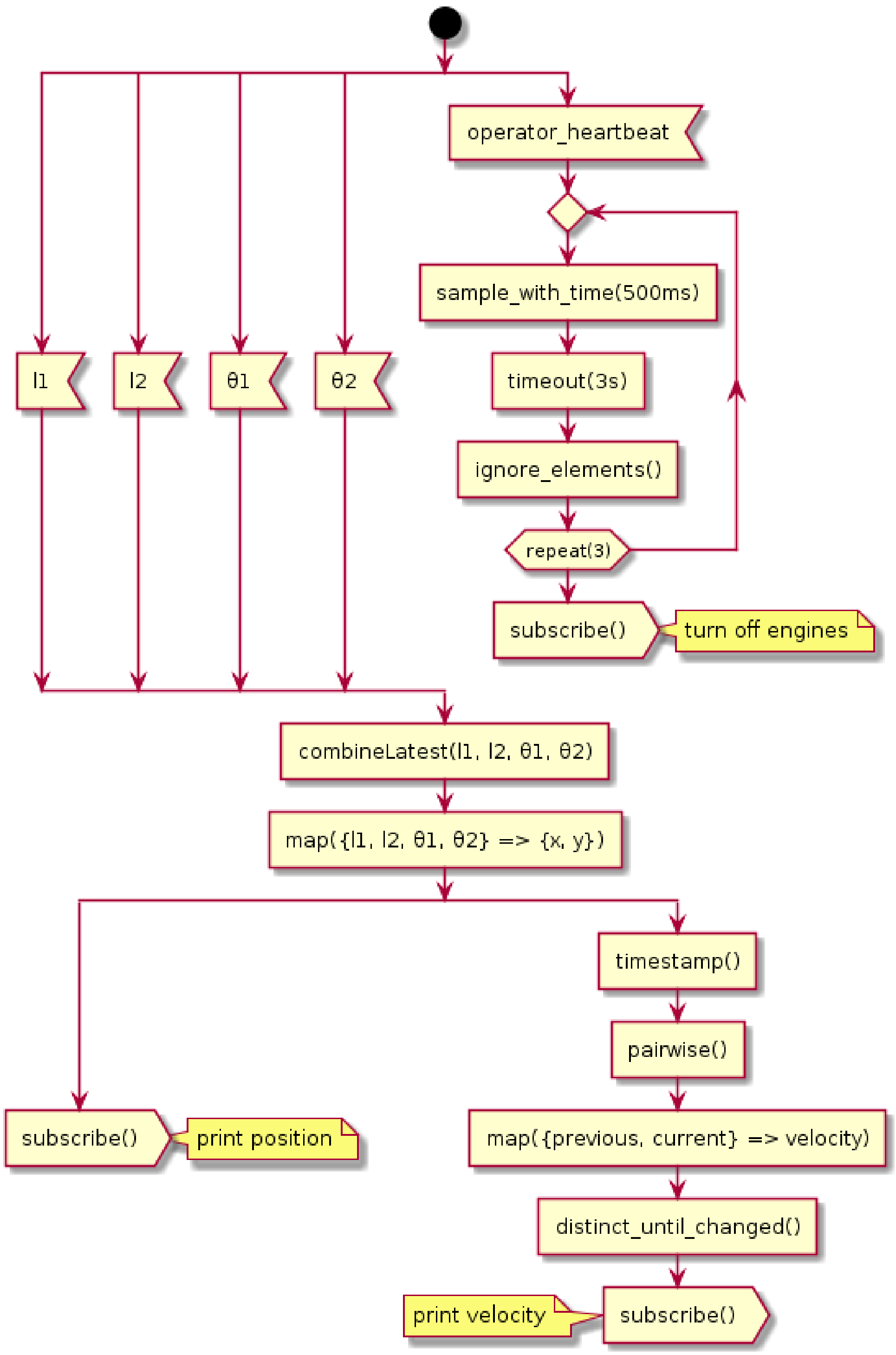


Figure 5.13 Summarized architecture of developed application.

Chapter 6

Summary

The main goal of this thesis was to verify application of functional and reactive programming concepts in distributed robotic control systems. In order to verify correctness of described composition, described in Chapter 4 library has been developed. Application of described concepts has been verified by implementation of exemplary robotic application (Chapter 5).

On the basis of obtained results it has been confirmed, that defined programming model can be used to solve various types of programming problems. As a first application area can be distinguished aggregation of information received from asynchronous resources. In particular case, logic of the information transferred in the streams can be aggregated with combining operators, which are concealing thread synchronization mechanisms from the user. Moreover, time critical operations can be performed in parallel by execution of reactive timers on independent threads. Finally has been observed, that error propagation over streams can be used to perform recovery operations and monitoring of the correct behaviour of the application inputs.

From the developer perspective, some advantages of proposed approach has been observed. Asynchronous stream model (represented by observables) allows for simplification of parallel programming process, because developer is defining relations between types instead of expressing sequential data flow. On the other side, logic of the application is realized with predefined and generic building blocks (operators), which can be used to operate on the streams in unified way. It is important to notice, that using of common functionalities makes code more readable, because all parts of the application are represented in similar way. In addition to this, code execution is performed without side effects, which simplify debugging procedures and general understanding of the code. What is more, on the basis of immutability property, developers can easily extend logic of the application without interference of it existing structure. Moreover, declarative approach allows to define relations between types and reusability of previously created relations. Described property has influence on testing procedure of defined recipes, because most of them are represented as a flow transforming some inputs into defined outputs in deterministic way. Into created in that way model can be injected testing informations, which should always be evaluated to the same results. It can be achieved, because time operations can be controlled by hidden in the framework mechanisms. Taking everything into account, proposed model allows for separation of thread management, time operations and error handling from business logic of the application.

Further considerations related to this topic can be focused on application of the continuous time model and usage of FRP paradigm. By modification of the time representation many robotic models can be expressed in declarative way, which with combination with proposed approach can allow to create common representation of the system logic.

Bibliography

[1] G. E. Moore, “Readings in computer architecture,” M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming More Components Onto Integrated Circuits, pp. 56–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=333067.333074

[2] “The law that’s not a law,” *IEEE Spectrum*, vol. 52, no. 4, pp. 38–57, April 2015.

[3] (2015) Netflix will re-encode its entire catalogue in 2016 to save bandwidth. [Online]. Available: https://thestack.com/cloud/2015/12/15/netflix-will-re-encode-its-entire-catalogue-in-2016-to-save-bandwidth/

[4] P. D. Francesco, “Architecting microservices,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, April 2017, pp. 224–229.

[5] M. Podwysocki. Reactive programming at netflix. [Online]. Available: https://www.youtube.com/watch?v=9fFWv4jmSgs

[6] Sandvine, “Global internet phenomena,” Report, December 2015. [Online]. Available: https://www.sandvine.com/hubfs/downloads/archive/2015-global-internet-phenomena-report-latin-america-and-north-america.pdf

[7] D. Brugali and P. Scandurra, “Component-based robotic engineering (part i),” *IEEE Robotics Automation Magazine*, vol. 16, no. 4, pp. 84–96, December 2009.

[8] D. Brugali and A. Shakhimardanov, “Component-based robotic engineering (part ii),” *IEEE Robotics Automation Magazine*, vol. 17, no. 1, pp. 100–112, March 2010.

[9] J. Backus, "History of programming languages i," R. L. Wexelblat, Ed. New York, NY, USA: ACM, 1981, ch. The History of Fortran I, II, and III, pp. 25–74. [Online]. Available: http://doi.acm.org/10.1145/800025.1198345

[10] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[11] S. Blackheath and A. Jones, *Functional Reactive Programming.* Manning Publications Company, 2016. [Online]. Available: https://books.google.pl/books?id=aO0zrgEACAAJ

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] R. Kurzweil, *How to create a mind : the secret of human thought revealed.* New York: Viking, 2012.

[14] M. Fowler and K. Scott, *UML Distilled (2Nd Ed.): A Brief Guide to the Standard Object Modeling Language.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[15] C. Elliott and P. Hudak, "Functional reactive animation," in *International Conference on Functional Programming*, 1997. [Online]. Available: http://conal.net/papers/icfp97/

[16] C. Elliott, *Why program with continuous time?*, 2010. [Online]. Available: http://conal.net/blog/posts/why-program-with-continuous-time

[17] D. S. Scott, "Outline of a mathematical theory of computation," Oxford University Computing Laboratory, Oxford, England, Technical Monograph PRG–2, November 1970.

[18] F. J. Pelletier, "The principle of semantic compositionality," *Topoi*, vol. 13, no. 1, pp. 11–24, Mar 1994. [Online]. Available: https://doi.org/10.1007/BF00763644

[19] J. Backus, "Can programming be liberated from the von neumann style?: A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359579

[20] *Means of expression.* [Online]. Available: https://wiki.haskell.org/Means_of_expression

[21] R. Machado, “An introduction to lambda calculus and functional programming,” in *2013 2nd Workshop-School on Theoretical Computer Science*, Oct 2013, pp. 26–33.

[22] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf

[23] R. H. B. Netzer and B. P. Miller, “What are race conditions?: Some issues and formalizations,” *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/130616.130623

[24] (2015) Synchronization quadrant. [Online]. Available: https://image.slidesharecdn.com/pythonadvanced-151127114045-lva1-app6891/95/python-advanced-building-on-the-foundation-156-638.jpg?cb=144891077

[25] (2018) Pure function. [Online]. Available: https://practicalli.github.io/clojure/images/functional-programming-concepts-pure-function.png

[26] C. Allen and J. Moronuki, *Haskell Programming from First Principles*. Allen and Moronuki Publishing, 2016. [Online]. Available: https://books.google.pl/books?id=5FaXDAEACAAJ

[27] D. Adams, *The Hitchhiker’s Guide to the Galaxy.* Pan Books, 1980.

[28] D. M. Ritchie and K. Thompson, “The unix time-sharing system,” *Commun. ACM*, vol. 17, no. 7, pp. 365–375, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361061

[29] T. H. Crowley, “Unix time-sharing system: Preface,” *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1897–1898, July 1978.

[30] G. Berry, “Real time programming : special purpose or general purpose languages,” INRIA, Research Report RR-1065, 1989. [Online]. Available: https://hal.inria.fr/inria-00075494

[31] J. Bonér, D. Farley, R. Kuhn, and M. Thompson. (2016) The Reactive Manifesto. [Online]. Available: https://www.reactivemanifesto.org/

[32] Reactive manifesto requirements diagram. [Online]. Available: https://www.reactivemanifesto.org/images/reactive-traits.svg

[33] Reactive filter operator. [Online]. Available: http://reactivex.io/documentation/operators/images/filter.png

[34] Reactive map operator. [Online]. Available: http://reactivex.io/documentation/operators/images/map.png

[35] Reactive combine latest operator. [Online]. Available: http://reactivex.io/documentation/operators/images/combineLatest.png

[36] Reactive debounce operator. [Online]. Available: http://reactivex.io/documentation/operators/images/debounce.png

[37] Reactive timeout operator. [Online]. Available: http://reactivex.io/documentation/operators/images/timeout.c.png

[38] Reactive catch operator. [Online]. Available: http://reactivex.io/documentation/operators/images/Catch.png

[39] J. Lewis and M. Fowler, *Characteristics of a Microservice Architecture*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[40] Y. Izrailevsky and A. Tseitlin, *The Netflix Simian Army.* Netflix Tech Blog, 2011. [Online]. Available: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116

[41] M. Bell, *Service-Oriented Modeling: Service Analysis, Design, and Architecture*, ser. NetLibrary, Inc. Wiley, 2008. [Online]. Available: https://books.google.pl/books?id=NAyhSRjPWEIC

[42] P. Education, *Agile Software Development, Principles, Patterns, and Practices.* Robert C. Martin, 2003.

[43] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 123–138, Nov. 1987. [Online]. Available: http://doi.acm.org/10.1145/37499.37515

[44] *Data Distribution Service.* Object Managment Group, 2015. [Online]. Available: https://www.omg.org/spec/DDS/1.4/

[45] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*, ser. Springer Handbook of Robotics. Springer Berlin Heidelberg, 2008. [Online]. Available: https://books.google.pl/books?id=Xpgi5gSuBxsC

[46] *DDS Interoperability Wire Protocol.* Object Managment Group, 2014. [Online]. Available: https://www.omg.org/spec/DDSI-RTPS/2.2/

[47] *Interface Definition Language.* Object Managment Group, 2018. [Online]. Available: https://www.omg.org/spec/IDL/4.2

[48] H. Pérez and J. J. Gutiérrez, “Modeling the qos parameters of dds for event-driven real-time applications,” *J. Syst. Softw.*, vol. 104, no. C, pp. 126–140, Jun. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.03.008

[49] “User Datagram Protocol,” RFC 768, Aug. 1980. [Online]. Available: https://rfc-editor.org/rfc/rfc768.txt

[50] R. Schollmeier, “A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications,” in *Proceedings First International Conference on Peer-to-Peer Computing*, Aug 2001, pp. 101–102.

[51] Open Source Robotics Foundation, *ROS.org | Powering the world’s robots.* [Online]. Available: http://www.ros.org/

[52] ——, *Understanding ROS Nodes.* [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes

[53] R. Thurlow, “RPC: Remote Procedure Call Protocol Specification Version 2,” RFC 5531 (Draft Standard), Internet Engineering Task Force, May 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5531.txt

[54] W. Woodall, *ROS on DDS.* [Online]. Available: https://design.ros2.org/articles/ros_on_dds.html

[55] ReactiveX library. [Online]. Available: http://reactivex.io

[56] E. Meijer, “Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues,” in *ACM SIGPLAN Commercial Users of Functional Programming*, ser. CUFP ’10. New York, NY, USA: ACM, 2010, pp. 11:1–11:1. [Online]. Available: http://doi.acm.org/10.1145/1900160.1900173

[57] Z. Wan, W. Taha, and P. Hudak, “Real-time frp,” *SIGPLAN Not.*, vol. 36, no. 10, pp. 146–156, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/507546.507654

[58] B. Stroustrup, *The C++ Programming Language.* Pearson Education, 2013. [Online]. Available: https://books.google.pl/books?id=PSUNAAAAQBAJ

[59] M. Cohn, *User Stories Applied: For Agile Software Development.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[60] (2001) Agile manifesto. [Online]. Available: http://agilemanifesto.org/

[61] Two link manipulator. [Online]. Available: https://i.stack.imgur.com/hfUPe.png

[62] (2014) Marble diagrams. [Online]. Available: http://rxmarbles.com/

[63] Reactive timestamp operator. [Online]. Available: http://reactivex.io/documentation/operators/images/timestamp.c.png

[64] Reactive distinct until changed operator. [Online]. Available: http://reactivex.io/documentation/operators/images/distinctUntilChanged.png

[65] Dead Man's switch. [Online]. Available: http://densorobotics.com/content/user_manuals/19/img/001704/001704_2.png

[66] Reactive throttle last operator. [Online]. Available: https://cdn-images-1.medium.com/max/1280/1*2SwHDsqskk0IS6v1T63dyw.png

[67] Reactive retry operator. [Online]. Available: http://reactivex.io/documentation/operators/images/retry.C.png

[68] Reactive ignore elements operator. [Online]. Available: http://reactivex.io/documentation/operators/images/ignoreElements.c.png

List of Figures

Listings

List of Tables