Prime
A Service-oriented Framework with Minimal Communication Overheads

Sergey Zhigalov[1] and Yuri Okulovsky[2]
[1]*Yandex N. V., Lev Tolstoy street, 16, 119021, Moscow, Russia*
[2]*Institute of Mathematics and Computer Science, Ural Federal University, Lenina 51, 620000, Yekaterinburg, Russia*

Keywords: Control Systems, Service-Oriented Approach, Framework.

Abstract: We present Prime, a framework for development of service-oriented control systems in robotics. Prime uses an original approach to services: the service is not a monolith, but is subdivided into three layers. This approach allows creating services' internal logic without references to communication-related entities, and therefore almost without initial learning of Prime. In addition, Prime offers three methods of linking services together that are completely interchangeable and compatible. The first is a classic service-oriented solution; the second belongs more to functional programming, it combines the algorithms inside each service into one function, that is equivalent to the behavior of the service-oriented system. The third uses code emission technique to significantly increase the performance. The Prime adds little overheads and is much faster than, for example, Microsoft Robotics Studio.

1 INTRODUCTION

The control systems for modern autonomous robots are incredibly complex. They contain dozens of algorithms for motion planning and control, processing data from sensors, etc. These algorithms need to be developed, tested and then combined together into one program. Managing this process is a non-trivial problem for a software architects. One of the approaches to the problem is called service-oriented (Kramer, Scheutz, 2007; Somby,2008). It demands each algorithm to be developed as a service, i.e. a separate entity that performs a single task and communicates with other services to obtain the input data or to pass the output data further. The most prominent implementations of this approach are Robot Operating System (ROS, (Foote et al., 2009)) and Microsoft Robotics Developer Studio (MRDS, (MRDS)).

Service-oriented approach brings many benefits. The development of the control system may easily be distributed between several programmers, and each of them should only consider his or her part of the job. The control system can evolve gradually: at the first iteration, the developers may create a draft system with services stubs that perform a very basic functionality; at later stages, they replace some services with the more complex ones. This facilitates AGILE development methodology in robotics, as well as the educational and research process: the development becomes the series of the small steps, and each of them is easy enough to understand and to try. Service-oriented systems are also more testable, since each algorithm can be separated from others and run with test data. The performance of service-oriented system may be easily increased by distributing it over several computers.

However, service-oriented approach has two major disadvantages. The first is the communication overheads. The framework for service-oriented programming requires time to pass data between services, and this time is considerable and often unpredictable. The second drawback is that the algorithms in the service-oriented control system become aware of the framework. They use framework-specific entities (like communication ports, or messages) to send and receive data. This brings the following consequences:

- All the developers have to learn about the framework. In some cases there is a lot to learn: for example, the complexity of MRDS is often considered as its biggest disadvantages.
- The services require the operating framework to consume data, and thus the framework should be set up for testing. It increases time of tests execution.

Zhigalov S. and Okulovsky Y..
Prime - A Service-oriented Framework with Minimal Communication Overheads.
DOI: 10.5220/0005017201690175
In *Proceedings of the 11th International Conference on Informatics in Control, Automation and Robotics* (ICINCO-2014), pages 169-175
ISBN: 978-989-758-039-0

- Since the algorithms usually have references to framework-specific parts, they are locked inside their frameworks. The only way to use them outside is to rewrite code, replacing these communication entities.

We describe our implementation of service-oriented approach to robotics. It has minimum communication overheads, and facilitates the development of services in comparison with ROS and MRDS. The approach is based on the original topology of the services, which was first described in (Kononchuk et al., 2001). Existing solutions use the classic service-oriented approach, when services can arbitrary call each other. To achieve that, Microsoft Robotics Studio uses a star topology, and ROS uses all-to-all broadcast. Our topology resembles the models from LabView (Travis,2001) or Simulink (Karris, 2008). The services cannot communicate arbitrarily: they have inputs and outputs, which are strictly interconnected, and so data flow only along these connections. In this article we show, how this topology may be exploited to obtain the best performance while keeping all the advantages of service-oriented approach, and present Prime - the middleware for robotics control system.

Our achievement required three steps. At the first one, we decompose the service-oriented system into three layers. Services themselves, as entities in service-oriented system, are placed at the Topology layer. The algorithms inside the services are separated into the Logic layer, and are completely unaware of SOA. The last Media layer defines what it means, to be a service and to interconnect them.

The second step was to create two completely different Media layers and thus two service-oriented frameworks, Optimus Prime and Liberty Prime. Optimus Prime uses the no-SQL database Redis (Redis) to interconnect services. Redis is a key-value storage and, roughly speaking, stores information in slots. The Optimus Prime service around a mathematical function is a thread that waits for the data appear in the slot, and then processes it with the function and stores in another slots. Services are interconnected by routing data in the slots, and the routing topology is set up by the Prime methods. So Optimus Prime is yet another service-oriented approach, with the usual communication overheads.

Liberty Prime is different. The user calls the very same methods to create a topology, and so may think that he or she creates the services and then links them together. But in fact nothing of that sort happens. Instead, Liberty Prime uses the functional programming approach: it combines the logics of the services into one big function that performs the same actions the original Optimus Prime topology did.

Finally, at the third step we improved Liberty Prime with code emission technique and obtained Radiant Prime, and thus achieved the best performance.

All the Media are completely interchangeable. Prime uses the factory pattern (Gamma et al., 1994) to create services from their logic and to interconnect them. To switch the Media we should only rewrite one line of code and create a different factory. Moreover, all Media are compatible, and we can create parts of the system with Liberty Prime and then interconnect them with Optimus Prime, hence distributing the system.

Prime is published at `gitbub.com/air-labs/Prime` under GPL v3 license, and is free to use and to extend. It is implemented in C# language and .NET framework (Drayton et al., 2002).

In section 1, we describe our approach in more details. In section 2 we describe our experience with Prime, and the section 3 bring the result of the computational experiment which certifies Prime outstanding performance.

2 PRIME ESSENTIALS

2.1 Linking Services

In this section, we describe the most fundamental feature of Prime: linking two services together. We explain this process in much technical details; however, they are crucial to understand how Prime works and how it achieves the goals we assigned in Introduction.

Suppose we have two algorithms, `FirstAlgorithm` and `SecondAlgorithm`, and want to link them together with Prime. Algorithms should be classes that implement `IFunctionalBlock` interface. The interface is defined at the Logical layer, and has the single `Process` method, which accepts an input and produces an output. So implementing algorithms for Prime does not require any special knowledge, the developer only needs to format the algorithm as a single function (which may, of course, call other functions, and do anything the .NET method can).

To handle algorithms inside Prime, we use `PrimeCore` class, a factory to create topology entities. The `PrimeCore` can convert `FirstAlgorithm` to `FirstChain` that is a topology entity. It implements `IChain` interface, which corresponds to a service with one input and one output. The exact meaning of “inputs” and “outputs” will be defined later.

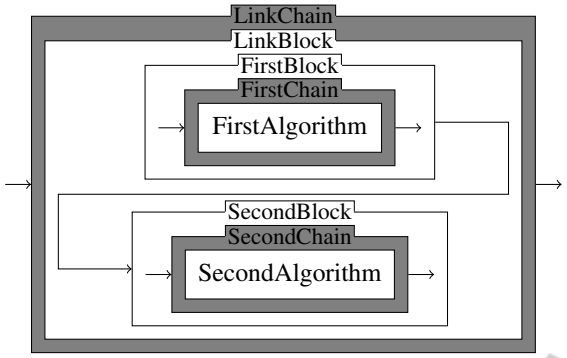

Figure 1: Layered diagram of linked services.

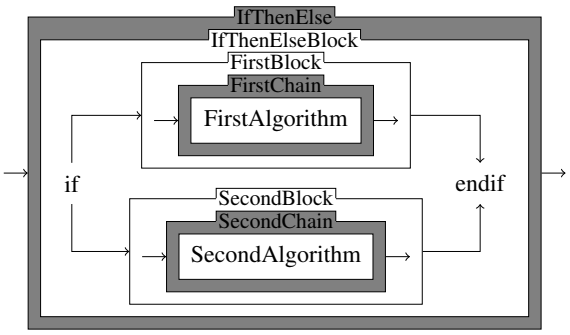

Figure 2: Layered diagram of `if` statement.

`PrimeCore` can convert `FirstChain` back to the Logic layer, creating a `FirstBlock` that is `IFunctionalBlock`. However, `FirstBlock` is not the `FirstAlgorithm`! The implementation of the `FirstBlock.Process` is to put a value in the `FirstChain` input, then wait for the result appears in the output, and return it. So `FirstBlock.Process` calls the `FirstAlgorithm.Process`, but indirectly.

To link `FirstChain` and the `SecondChain` together, they should be converted to `FirstBlock` and `SecondBlock`. Then we create a `LinkBlock` instance. `LinkBlock` implements `IFunctionalBlock`, it keeps references to `FirstBlock` and `SecondBlock`, and its Process method is defined as follows:

```
public TOut Process(TIn input) {
return SecondBlock.Process
    (FirstBlock.Process(input));
}
```

Since `LinkBlock` is `IFunctionalBlock`, `PrimeCore` may create `LinkChain` out of it, which is the result of linking `FirstChain` and `SecondChain`.

It may seem redundant to convert `FirstAlgorithm` to `FirstChain` and then to `FirstBlock`. However, `FirstChain` could be a composite chain itself (like `LinkChain`), and in this case its conversion to `IFunctionalBlock` is necessary, so we brought a general algorithm.

The Fig. 1 represents the result of the linking procedures. Here gray rectangles denote Topology entities, while white rectangles correspond to the Logic layer. Each transition between Logic and Topology layer requires methods on the Media layer.

Note that despite both chains are at the Topology layer, the logic of their interaction is placed inside `LinkBlock`, a logical entity, and hence unaware of topology. So we may easily invent other ways to combine chains: for example, `IfBlock` to reroute the input to the `FirstChain` or the `SecondChain` according to a given condition (see Fig. 2). The same approach was used to create loops and others standard programming concepts. Potentially, with Prime one may assemble an algorithm from blocks, like in Lab View or Simulink.

2.2 Media layer

All versions of Prime offer different implementations of `IChain`. `OptimusChain` is a service in the usual sense of this word. It keeps the names of Redis's `InputSlot` and `OutputSlot`. When run, it enters the infinite loop in the separated thread, waits for the value in Redis with the key `InputSlot`, then reads and deserializes it, calls the `Process` method of the inner `IFunctionalBlock`, and stores the output in Redis with the `OutputName` key.

`LibertyChain` is completely different. It is not a service; it only has a pointer `Function`, which references to the `Process` method of the `IFunctionalBlock` inside. The Fig. 3.a shows the complete map of methods inside the `LinkChain` that is built by Liberty Prime.

The main advantage of Prime is an ability to switch from the service-oriented to the functional programming paradigm and therefore minimize the overheads. To perform the switch, the developer should only replace OptimusCore with LibertyCore. Since they are implementations of the same abstract PrimeCore class, no further code revisions are required. Media can be used together: in examples on Fig. 1 and 2, `FirstChain` and `SecondChain` may be Optimus Prime chains and run on remote computers, while `LinkChain` or `IfThenElseChain` remains a local Liberty Prime chain.

The linking algorithm from the paragraph 1.1 is actually never used. All the media improve it. `OptimusCore` just reassigns `SecondChain.InputName` to `FirstChain.OutputName`, and returns `LinkChain` such that `LinkChain.InputName` equals to `FirstChain.InputName`, and `LinkChain.OutputName` equals to

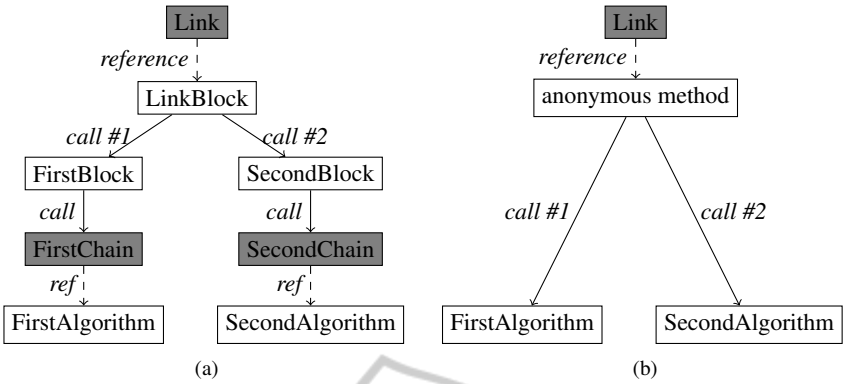

Figure 3: Initial (a) and optimized (b) map of methods in services' link.

`SecondChain.OutputName`. When `LibertyCore` links chains, it assigns `LinkChain.Function` to the composition of `FirstChain.Process` and `SecondChain.Process` (see Fig. 3.b for the revised methods map). Therefore, ideally the overheads in Liberty Prime are reduced to the costs of the two methods invocations.

Liberty Prime approach can be implemented in any object-oriented language. However, in .NET framework the synchronous linking may be further improved with the code emission technique and LINQ Expressions library. Code emission allows writing code in the runtime, compiling it, and then running with the performance of the code that is written in the traditional way. LINQ Expression simplifies this process, offering the expressions tree, which are the object-oriented representation of an ordinary syntax tree.

We developed a third version of Prime, Radiant Prime. `RadiantChain` keeps a link to the expression that corresponds to the chain. For example, `FirstChain` stores the expression `x=>firstBlock.Process(x)`, and the second stores `y=>secondBlock.Process(y)`. These expressions are not delegates, are not the pointers functions, they are syntax trees for corresponding functions. When chains are concatenated, we substitute `y` in the `SecondChain` with `firstBlock.Process(x)`, thus obtaining the resulting function. When `ToFunctionalBlock` is called, the expression is compiled and transformed into a delegate. This delegate then used to process data with `P`rocess method.

Radiant Prime can also build expressions for `IfThenElse` of `Foreach` chains, as well as for other synchronous actions.

2.3 Asynchronous Data Processing

Let us briefly describe asynchronous data processing with Prime. The first logical entity is an `IEventBlock`, an interface with the single defined `Event`. At the Topology layer, `ISource` is defined, an entity that is analogous to `IChain`, but with one output only. `PrimeCore` creates `ISource` out of `IEventBlocks`, and when `IEventBlock.Event` is raised with some value, this value is stored in Redis slot (in Optimus Prime) or in a buffer (in Liberty and Radiant Prime). `ISource` can be then converted to `IReader`, a logical entity, which is capable to read values from output: one value at a time, or all the published values as an array.

To link `IChain` to `ISource`, we use the same approach as to synchronous linking. We introduce the `SourceLinkBlock` (its layered diagram is represented at the Fig. 4.a.) When `ISourceLinkBlock` starts, it enters the infinite loop at a separate thread, processes the values from `IReader` with a `Block`, and publishes them in `EventBlock`. Note that the combination of `ISource` and `IChain` is `ISource`, and additional chains may be then linked to it.

Prime allows managing threads when working with asynchronous data. For example, if chains are combined into one chain and then linked to `ISource`, only one thread for the assembled chain will be created. In addition, data from sources can be synchronized with the aid of `Collector`. The `Collector` is a chain, it contains `CollectorBlock` with `IReader` inside, and returns the data from `IReader` in response to an empty object `Token` (see Fig. 4.b for a layered diagram). It adds to thread, but data from the source will be processed synchronously.

Radiant Prime currently does not differ from Liberty Prime in asynchronous data processing.

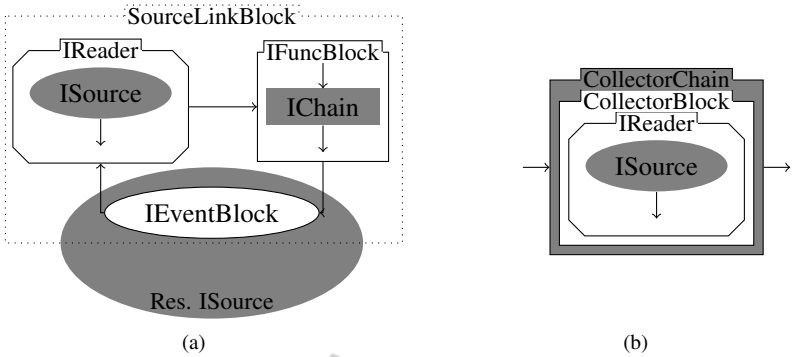

Figure 4: Layered diagram (a) of a chain, linked to a source; (b) of a collector.

3 EXPERIENCE OF USING PRIME

In our laboratory we used Prime and its prototype for two years as a primary tool to develop control systems for Eurobot competitions (Eurobot). We developed numerous auxiliary services (like `Collector` or `IfThenElse`, and two working control systems for a differential wheeled robot and a manipulator, consisting of more than 50 services in total.

The main advantage Prime brings into the development process is a quick start. When a student comes to the laboratory and shows interest to robotics, it is important (by our opinion) to instantly provide him or her the opportunity to make something useful, right now. It was not possible with MRDS or RoboCoP, the systems we used in the past. They required too much knowledge even to start working with, and students experienced a decrease of motivation and excitement when told so.

With Prime, however, we may instantly assign student to some algorithm, because the only thing he or she needs to do is to encode algorithm as the implementation of `IFunctionalBlock`. The code may be further tested by unit testing, or even created under Test-Driver Development methodology, when we write tests at the first place, and only then try to implement the code that passes them. It brings the development of robotics control systems closer to a pure software development, without robotics peculiarities (except for those that are in the control algorithms themselves). And it is important in our environment, because the laboratory is situated at the Institute for Mathematics and Computer Sciences, and so the priority is to teach algorithms and software development, and not the features of MRDS or ROS.

After the algorithm is developed and tested, it may be included to a control system. This requires a next level of understanding Prime: how one should connect services to make them work. However, it is easy to explain, and diagrams of services (the usual ones, not the layered diagrams from this article) are effective even for pupils that are 12-14 years old: for example, Lego Mindstorms and Aldebaran NAO robots provide them. Prime lacks the visual system for editing topologies (and we are not planning to add it, because Prime is designed for the software development students, not the pupils), but methods of `PrimeCore` reflects the diagrams very well.

The complete understanding of Prime layers is required only to develop the Media or auxiliary logical blocks, like `Collector` or `IfThenElse`. However, such tasks are extremely rare, so we do not have teach all the students to perform them, we need only a few students, who work in the robotics for a long time and thus are motivated enough to learn about Prime and have enough time to do that.

Let us also note that Liberty Prime uses elements of functional programming paradigm and asynchronous tasks management. By our observations, both these topics are traditionally hard for computer science students. However, Liberty and Radiant Prime effectively hide them behind the mask of service-oriented approach. Hence, Liberty and Radiant Prime have the potential to become a framework for other software systems, not only in robotics.

In addition, Prime brings all the standard improvements, associated with service-oriented approach: the development becomes less centralized, and students may develop tasks independently, in different branches of version control systems, merging and working together only when their task is completed. Prime also supports AGILE software development methodology, by gradual replacement services with their new, more effective versions, and thus allows us using the practiced that are common for software development.

Table 1: Performance comparison for several chains, in asynchronous and synchronous modes, in microseconds per transmission.

Synchronouse mode

	Small data	Medium data	Big data
FOR	$0,024 \pm 0,001$	$0,017 \pm 0$	$0,017 \pm 0$
Radiant Chains	$0,031 \pm 0,001$	$0,024 \pm 0$	$0,023 \pm 0,001$
Liberty Chains	$0,335 \pm 0,006$	$0,325 \pm 0,005$	$0,323 \pm 0,006$
CCR (in sync mode)	$5,687 \pm 0,518$	$5,194 \pm 0,648$	$5,169 \pm 0,579$
Prime Sources	$0,642 \pm 0,022$	$0,498 \pm 0,009$	$0,491 \pm 0,012$
CCR (in async mode)	$1,254 \pm 0,025$	$0,949 \pm 0,02$	$0,913 \pm 0,019$

Asynchronouse mode

	Small data	Medium data	Big data
FOR	$0,018 \pm 0$	$0,006 \pm 0$	$0,005 \pm 0$
Radiant Chains	$0,018 \pm 0$	$0,007 \pm 0$	$0,007 \pm 0$
Liberty Chains	$0,184 \pm 0,008$	$0,131 \pm 0,006$	$0,125 \pm 0,004$
CCR (in sync mode)	$8,654 \pm 0,493$	$8,379 \pm 0,47$	$8,402 \pm 0,405$
Prime Sources	$0,954 \pm 0,046$	$0,743 \pm 0,026$	$0,726 \pm 0,032$
CCR (in async mode)	$0,918 \pm 0,018$	$0,812 \pm 0,025$	$0,783 \pm 0,018$

4 PERFORMANCE OF PRIME

In this section, we describe the computational experiment that certifies the effectiveness of Prime. Our main objective was to test the approach rather than implementation, and so we choose to compare Liberty and Radiant Prime to Concurrency and Coordination Runtime (CCR), a Microsoft .NET solution to messaging in service-oriented robotics system. We did not considered Decentralized Software Services technologies, which is built upon CCR and is a true core of Microsoft Robotics Studio, because DSS adds overheads of interprocess interaction that Liberty Prime does not have. We have not considered Optimus Prime, because the majority of overheads are introduced by Redis in this case, and testing Redis performance is not a theme of our article; in addition, we do not declare that Redis is the best solution for interprocess communication, and Prime architecture allows us replacing it with another technology easily. We have also not tested ROS, because it runs on different framework and on different operating system, so it would be unclear, whether overheads are brought by communication approach or by framework; however, based on (Hongslo, 2012), we assume ROS performance is inferior to the CCR.

For experiment, we used a topology that consists of several chains of services. Each service performs an addition function, and therefore consumes a negligible time. The data that flow through the services, and can be small (100 bytes), medium (1 kilobyte) or big (10 kilobytes). There are two modes of how each chain processes signals:

- Asynchronous mode, when data flow through the chain constantly. The common example is processing data from sensors. It is the native behavior of CCR. In Prime, it is represented as `ISource` with the subsequently connected `IChain` (let us call this topology Prime Source for shortness).
- Synchronous mode, when the new input signal appears only when the output signal is produced for the previous input, most commonly, as a response to it. Synchronous chains are the most common element of our control systems. The systems are mostly synchronous; however, they sometimes require the collection of processed asynchronous data. Prime Sources and CCR may operate in this mode, if they are augmented with a dispatcher that releases an input signal when the output is computed. The presumed implementation of this pattern in Prime is linked chains, which have different implementation in Radiant and Liberty Prime.

For comparison, we also consider the time, requried to achieve the same task without any services, by calling the methods directly inside `for` cycle. This "system" is represented in results with the name FOR.

At the first series of our experiment, we directly measured overheads in Prime Sources, Prime Chains and CCR. Two later systems were forced to work in synchronous mode by the dispatcher. We measured the total time of processing several signals, and obtained the overheads time per one transmission, which is shown in Table 1 for various package sizes.

The time depends on packages sizes only slightly, because neither Prime not CCR does not serialize or deserialize them for local systems. We can see that

Liberty Chains is about 15 times faster than CCR, and Radiant Chains are about 10 times faster than that. The performance of Radiant Chains is only slightly worse than in FOR system. We consider it as a confirmation that both our approach and its implementation are successful: we managed to create a service-oriented framework with minimal overheads.

At the second series, we explored the asynchronous case. Eight chains was run to process signals; in synchronous mode, the chains worked in parallel, but each chain still processed the signals synchronously. The experiment was performed on a computer with 8 cores, so interaction between threads brings additional overheads. The load of all cores during the experiment was close to 100%.

We may see the unusually low performance of CCR in synchronous mode. Perhaps, the reason for that is that the synchronous mode is not native for CCR, and therefore CCR is not optimized for that. Prime Chains processes signals even faster, that CCR in asynchronous mode, in spite the Prime chain processes only one signal in a time, and CCR in asynchronous mode processes several.

Prime Source sometimes has worse performance that CCR. However, we know ways to optimize them, and in a real practice we never build systems of that architecture. Instead of linking several chains to a source, we link these chains together, obtain a synchronous chain and then link it to a source, achieving the performance of Prime Chains.

5 CONCLUSION

In this paper, we presented Prime, a framework for development of service-oriented control system. We described layered model or service-oriented software, and explained how it is applied for synchronous control systems. We demonstrated an essential performance improvement in this synchronous case. The approach was used in creation of two control systems for real robots, and we believe it is much easier to use and to learn.

We understand that Prime is yet not ready for the industrial exploitation. Also, Prime is implemented for .NET framework in C# language, which is probably not the best selection of the industrial control system. However, we believe the approach we offer is useful even for the real-time and industrial systems, because one can precisely compute the amount of time, required for linking services in synchronous mode.

ACKNOWLEDGEMENTS

The work is supported under the Agreement 02.A03.21.0006 of 27.08.2013 between the Ministry of Education and Science of the Russian Federation and Ural Federal University.

We would like to thank Alexander Mangin for criticism and valuable comments that help us make Prime faster, better and much more understandable.

REFERENCES

Drayton, P., Albahari, B., Neward, T. C# in a Nutshell. O'Reilly, Sebastopol (2002)

Eurobot association, http://eurobot.org.

Foote, J. L., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source Robot Operating System (2009), http://www.robotics.stanford.edu/ang/papers/icraoss09-ROS.pdf

Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Hongslo, A.: Stream Processing in the Robot Operating System framework (2012), http://www.ida.liu.se/divisions/aiics/publications/Exjob-2012-Stream-Processing-Robot.pdf

Karris, S. T.: Introduction to Simulink with Engineering Applications, 2e. Orchand Publications (2008).

Kononchuk D.O., Kandoba V.I., Zhigalov S.A., Abduramanov P.Y., Okulovsky Y.S.: RoboCoP: a Protocol for Service-Oriented Robot Control System. Proceedings of international conference on Research and Education in Robotics - Eurobot (2011).

Kramer, J., Scheutz, M.: Development environments for autonomous mobile robots: A survey. In: Autonomous Robots, V. 22. pp. 132 (2007)

Microsoft Robotics Developer Studio. http://msdn.microsoft.com/en-us/robotics/default.aspx

Mono project. http://www.mono-project.com

Redis homepage. http://redis.io

Robotics operating system. http://www.ros.org

Somby, M.: Software Platforms for Service Robotics (2008) http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Updated-review-of-robotics-software-platforms/

Travis, J. LabVIEW for Everyone. Prentice Hall, Upper Saddle River (2001)