GNSS-SDR: an Open Source Tool For Researchers and Developers

Carles Fernández–Prades, Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Spain.
Javier Arribas, Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Spain.
Pau Closas, Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Spain.
Carlos Avilés, Carlos Avilés Software (CAS), Germany.
Luis Esteve, Universitat Politècnica de Catalunya (UPC), Spain.

BIOGRAPHY

Dr. Carles Fernández–Prades is a Research Associate at the Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), where he holds a position as a Coordinator of the Communications Subsystems Area. He received the PhD degree from Universitat Politècnica de Catalunya (UPC) in 2006. His primary areas of interest include signal processing, estimation theory, GNSS synchronization, and the design of RF front-ends.

Mr. Javier Arribas is a PhD Candidate at the Centre Tecnològic de Telecomunicacions de Catalunya (CTTC). He received the BSc and MSc degree in Telecommunication Engineering in 2002 and 2004 respectively at La Salle University in Barcelona, Spain. He is currently involved in the SalleSat Cubesat picosatellite development project. His primary areas of interest include statistical signal processing, GNSS synchronization, estimation theory and the design of RF front-ends.

Dr. Pau Closas received the MSc and PhD degrees in Electrical Engineering from the Universitat Politècnica de Catalunya (UPC) in 2003 and 2009, respectively. Currently he holds a position as a Research Associate at CTTC within the Communications Subsystems Area. His primary areas of interest include estimation theory, GNSS synchronization, Bayesian filtering and robustness analysis.

Mr. Carlos Avilés is a Software Engineer and develops his professional activities mainly in Berlin, Germany. He has received a Computer Science Engineering degree from Universitat Oberta de Catalunya (UOC) in 2011. His Diploma Thesis was developed as a collaboration with CTTC within the GNSS-SDR project.

Mr. Luis Esteve is a Telecommunications Engineering student at UPC. He is working on his MSc Thesis in collaboration with CTTC within the GNSS-SDR project. He is teacher and co-owner at Epsilon Formación SL since 1999.

ABSTRACT

This paper introduces GNSS-SDR, an open source Global Navigation Satellite System software-defined receiver. The lack of reconfigurability of current commercial-of-the-shelf receivers and the advent of new radionavigation signals and systems make software receivers an appealing approach to design new architectures and signal processing algorithms. With the aim of exploring the full potential of this forthcoming scenario with a plurality of new signal structures and frequency bands available for positioning, this paper describes the software architecture design and provides details about its implementation, targeting a multiband, multisystem GNSS receiver. The result is a testbed for GNSS signal processing that allows any kind of customization, including interchangeability of signal sources, signal processing algorithms, interoperability with other systems, output formats, and the offering of interfaces to all the intermediate signals, parameters and variables. The source code release under the GNU General Public License (GPL) secures practical usability, inspection, and continuous improvement by the research community, allowing the discussion based on tangible code and the analysis of results obtained with real signals. The source code is complemented by a development ecosystem, consisting of a website (`www.gnss-sdr.org`), as well as a revision control system, instructions for users and developers, and communication tools.

1. INTRODUCTION

Location has become an embedded feature not only on medium and high-end mobile phones, but also on other portable devices such as digital cameras and portable gaming consoles. This massive deployment of GNSS receivers requires a high level of integration, a low cost, a small size and a low power consumption, which has pushed the leading GPS integrated circuit (IC) manufacturers such as Qualcomm Inc., Broadcom Corporation, Cambridge Silicon Radio (CSR, merged with SiRF in 2009), Texas Instruments Inc., STMicroelectronics, u-Blox AG, Maxim or MediaTek to offer single-chip solutions easy to integrate in multi-function devices. Thus, the radio frequency (RF) front-end and the baseband processing are jointly implemented in monolithic ICs, tiny black boxes leaving the user no possibility to interact or to modify the internal architecture or the algorithmics. This approach is very convenient for location based services and applications, since users and developers are interested in *using* the location information (eventually taking advantage of complementary information coming from wireless network providers) but not in *how* the position has been obtained. Good examples of this abstraction can be found in the application programming interfaces (APIs) of two major operating systems for mobile devices: Apple's iOS provides a core location framework with objects that incorporate the geographical coordinates and altitude of the device's location along with values indicating the accuracy of the measurements, when those measurements were made, and information about the speed and heading in which the device is moving. A similar situation is found in Android, which provides a location package that contains classes with descriptive-named methods such as getLatitude(), getLongitude(), getAltitude(), getSpeed(), getAccuracy() and so on. This abstraction layer simplifies a lot the job of the application developer, but leaves no way to observe or modify any internal aspect of the receiver.

As an opposite driving force, the advent of a number of new GNSS (Galileo, COMPASS), the modernization of existing ones (GPS L2C and L5, GLONASS L3OC) and the deployment of augmentation systems (both satellite-based, such as WAAS in the USA, EGNOS in Europe, and MSAS in Japan; and ground-based, such as WiFi positioning and Assisted GNSS provided by cellular networks) depict an unprecedented landscape for receiver designers [1]. In the forthcoming years, many new signals, systems and frequency bands will be available for civil use, and their full exploitation will require a thoughtful redesign of the receiver's architecture and inner algorithms. In addition to being black boxes hidden by an abstraction layer, current mass-market GPS ICs are clearly constrained in terms of configurability, flexibility and capacity to be upgraded. This fact has headed receivers' designers to the software radio paradigm, in which an analog front-end performs the RF to intermediate frequency (or directly to baseband) conversion prior to the analog-to-digital converter (ADC). All remaining signal and data processing, including the hybridization with other systems, are defined in the software domain. This approach provides the designers with a high degree of flexibility, allowing full access and possibility of modification in the whole receiver chain.

The last decade has witnessed a rapid evolution of GNSS software receivers. Since the first GPS Standard Positioning Service software receiver described in [2], where the concept of bandpass sampling (or intentional aliasing) was introduced, several works were devoted to architectural and implementation aspects [3, 4, 5, 6, 7, 8, 9, 10, 11]. Textbooks [12] and [13] increased the awareness of the community about the great benefits provided by software receivers with respect to the traditional hardware-oriented approach, providing Matlab implementations of a complete GPS receiver, and [14, 15] provide discussions about high-level architecture design. In [16], authors presented an analysis of software design patterns and their application to GNSS software receivers.

Today, there are solutions available at academic and commercial levels, usually not only including programming solutions but also the development of dedicated RF front-ends. As examples, we can mention the GSNRx (GNSS Software Navigation Receiver [14]) developed by the Position, Location And Navigation (PLAN) Group of the University of Calgary; the ipexSR, a multi-frequency (GPS C/A and L2C, EGNOS and GIOVE-A E1-E5a) software receiver developed by the Institute of Geodesy and Navigation at the University FAF Munich [17, 18]; or N-Gene, a fully software receiver developed by the Istituto Superiore Mario Boella (ISMB) and Politecnico di Torino that is able to process in real time the GPS and Galileo signals broadcast on the L1/E1 bands, as well as to demodulate the differential corrections broadcast on the same frequency by the EGNOS system. This receiver is able to process in real-time more than 12 channels, using a sampling frequency of approximately 17.5 MHz with 8 bits per sample [19].

In this paper, we focus on *signal* processing, understood as the process between the ADC and the computation of code and phase observables, including the demodulation of the navigation message. We purposely omit *data* processing, understood as the computation of the navigation solution from the observables and the navigation message, since there are a number of well-established libraries and applications for that (also in the open source side, such as GPSTk [20, 21]). New available signals pose the challenge of multisystem, multiband receivers' design, including issues such as interference countermeasures, high-precision positioning for the mass-market, assisted GNSS and tight hybridization

with other technologies. In this context, this paper introduces an open-source GNSS software defined receiver (so-named GNSS-SDR) released under the GNU General Public License (GPL), thus ensuring the freedom of modifying, sharing, and using the code for any purpose. This secures practical usability, inspection, and continuous improvement by the research community, allowing the discussion based on tangible code and the analysis of results obtained with real signals. Hence, it is also intended to be a framework for algorithm testing and an educational tool, since everybody is allowed to peruse the source code, see how the receiver is actually implemented, and contribute with improvements, bug fixes, and addition of new features.

The remainder of the paper is as follows: Section 2 sketches the main characteristics and goals of the proposed software receiver, identifying possible signal sources, briefly describing the overall receiver architecture and design patterns, and analyzing which are the most useful formats for data output. Then, Section 3 provides architectural and mechanistic details about the different abstraction layers, from sample flow management to the actual implementation of signal processing algorithms. The Section also describes important features beyond the receiver's source code, such as the development ecosystem, quality assurance, or how we addressed portability. Finally, Section 4 concludes the paper.

2. RECEIVER'S OVERVIEW

The proposed receiver provides an interface to different suitable RF front-ends and implements all the receiver chain up to the navigation solution. Its design allows any kind of customization, including interchangeability of signal sources, signal processing algorithms, interoperability with other systems, output formats, and offers interfaces to all the intermediate signals, parameters and variables. The goal is to write efficient and truly reusable code, easy to read and maintain, with fewer bugs, and producing highly optimized executables in a variety of hardware platforms and operating systems. In that sense, the challenge consists of defining a gentle balance within level of abstraction and performance. The proposed software receiver runs in a commodity personal computer and provides interfaces through USB and Ethernet buses to a variety of either commercially available or custom-made RF front-ends, adapting the processing algorithms to different sampling frequencies, intermediate frequencies and sample resolutions. This makes possible rapid prototyping of specific receivers intended, for instance, to geodetic applications, observation of the ionospheric impact on navigation signals, GNSS reflectometry, signal quality monitoring, or carrier-phase based navigation techniques. Testing is conducted both by the systematic functional validation of every single software block (following a test-driven developing approach and using unit testing as a verification and validation methodology), and by experimental validation of the complete receiver using both real and synthetic signals.

2.1. Signal sources

An appealing feature for a software receiver is the possibility of working in real-time with real signals, when the processor is fast enough, or in an offline mode (post-processing) working with raw signal samples stored in a file, when the complexity of the implementation prevents from a real-time processing. Signals might also need to be created by synthetic signal generators in order to conduct experiments with controlled parameters.

Ideally, an all-software receiver should perform digitization right after the antenna. Due to technological constraints, there is still the need for amplification and down-conversion before the ADC, the so-called RF front-end. We also need an interface between the ADC output and the PC (or other general-purpose processor) in which the software receiver is running. This "hardware portion" of the receiver can be implemented with commercial off-the-shelf components or taking advantage of existing RF application-specific ICs. Modern ones feature single-conversion GNSS receivers, including the low noise amplifier (LNA) and mixer, followed by the image-rejected filter, programmable gain amplifier (PGA), voltage-controlled oscillator (VCO), frequency synthesizer, crystal oscillator, and a multibit (usually up to 3 bits) ADC equipped with an automatic gain control (AGC) system.

There are several signal grabbers commercially available. For instance, the Universal Software Radio Peripheral (USRP) [22] is a general-purpose family of computer-hosted hardware for software radios that, equipped with a DBSRX daughterboard [23] that can be used as a customizable RF front-end for GNSS receivers. Other more-specific, lower cost solutions are usually composed of an antenna, a RF IC front-end, a complex programmable logic device (CPLD) that arranges sample bits in bytes, and a USB 2.0 microcontroller. This is the case of the SiGe GN3S Sampler v2, based on the SiGe 4120 GPS IC [24], that provides a data stream with a sampling frequency of 16.3676 MHz and a bandwidth up to 4.4 MHz; the NSL's Primo (based on the Maxim's MAX2769 RF IC front-end [25], that allows to configure a bandwidth of 2.5, 4.2, 8, or 18 MHz and has a sampling frequency of up to 40 MHz) and Primo II (same characteristics but dual band, using a couple of MAX2769); and the IFEN's NavPort-III, a RF front-end able to work simultaneously in 4 frequency bands, with a bandwidth of 13 MHz each [26].

The state-of-the-art RF IC developments for GNSS receivers in 2011 focus on offering small size (25 mm^2 is a typical footprint), low power consumption (around 20 mA at 3 V), broader bandwidths than the commonly encountered 2 MHz

for GPS L1-only receivers in order to allow for the proper reception of GLONASS, Galileo, and COMPASS signals, low noise figure (around 1.6 dB for the cascaded chain) and low cost (about $5 US for a single unit). The trend is to provide a programmable IF frequency and to eliminate the need for external IF filters, allowing low-cost GNSS receiver solutions that require only a few external components. As examples, we can mention STMicroelectronics' implementation in CMOS 65 nm technology [27] and reports of dual-band front-ends integrated in a single chip [28, 29].

In summary, we identified two main requirements for the software receiver in terms of signal sources:

- It should allow real-time (when possible) and offline operation.
- It should be able to use a variety of signal sources (files and RF front-ends).

2.2. Receiver architecture

The software architecture has to resolve design forces that sometimes can be antithetical, such as flexibility vs robustness, or portability vs efficiency. With the objective of attaining real-time in mind, efficiency should be addressed specially in those blocks that work with high data rates (mainly, signal conditioning, Doppler removal, and correlation), while other blocks working at medium rate (tracking, extraction of navigation parameters) or low rate (measurement generation, navigation solution) can be implemented targeting robustness and reliability.

The chosen programming language is C++. The rationale not only relies on the fact that C++ is a dominant language, but also that it makes it much easier to recruit experienced programmers, and a number of well-written, peer-reviewed libraries (a crucial aspect in avoiding the reinvention of the wheel) and advanced compilers are available. Although it allows *close to the metal* programming (thus addressing efficiency), C++ also adds layers of abstraction that make possible the use of templates (i.e., functions and classes written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters). Templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled. This mechanism, known as static polymorphism, along with being a way of pre-evaluating some of the code at compile-time rather than at run-time, also reverts in more optimized machine code, smaller executables, shorter run-times, and lesser memory requirements, avoiding the overhead of run-time polymorphism. Moreover, a new version of the language, named C++11, has been recently approved as an ISO (International Organization of Standardization) standard [30]. This ensures that decades from now, today's standard conforming C++ programs will run with minimal modifications, just as an older C++ programs do today. It also ensures portability and the availability of compilers. This new version provides facilities for writing concurrent code (e.g. for multicore machines) in a type safe-manner, smart pointers, new memory-optimized ways of object handling, and tons of new core and library features that are very convenient for software radio applications.

The receiver's general block diagram is depicted in Fig. 1. It consists of a Control Plane in charge of managing the whole receiver and the interactions with the underlying operating system, external applications and user-machine interface; and a Signal Processing Plane that extracts information from raw signal samples.

The nature of a GNSS receiver imposes some requirements in the architecture design. Since the composition of the received GNSS signals will change over time (initially, some satellites will be visible, and after a while, some satellites will not be visible anymore and new ones will show up), some channels will loose track of their signals and some new channels will have to be instantiated to process the new signals. This means that the receiver must be able to activate and deactivate the channels dynamically, and it also needs to detect these changes during runtime.

There is also a need to design the communication mechanism within the signal processing blocks and the Control Plane. The Message Queueing pattern [31] approach is very flexible and can be adapted to almost any control we need to implement. There is a control thread that runs in parallel to the flowgraph, receiving notifications that trigger changes in the application. Some of these notifications will be sent directly from the processing blocks. For instance, an acquisition block that finishes its processing and detects a satellite's signal, will send a notification to the control thread via a message queue indicating its success. The control thread will then change the internal configuration of the channel and pass the results of the acquisition process (i.e., detection of in-view satellite and rough estimations of its code delay and Doppler shift) to the tracking blocks. Since GNSS-SDR is a multithreaded application, the control thread will receive messages from many sources and probably at the same time. The mechanism for sending and reading these messages must guarantee that only one thread reads or writes the shared data at a time, a condition known as *thread-safety*. The control thread requires a thread-safe queue; it shares an instance of this queue with all other modules that require sending messages.

Signal processing blocks process the input data stream, concurrently applying the same set of operational transformations, such as acquiring or tracking the signal of different

satellites. Hence, it is desirable an structure that improves capacity (i.e., the number of satellites to be tracked) with the replication of architectural units, allowing efficient parallel processing of data. The solution follows the Channel Architecture pattern [31], grouping all the signal processing related to a single satellite into a channel subsystem. A channel can be thought of as a pipe that sequentially transforms data from input values to output values, possibly at a different rate. The Channel Architecture pattern is well suited to the sequential transformation of data from one state or form to another. It simplifies algorithms that can easily be decomposed into a series of steps operating on isolate elements from a data stream. Instances of channel subsystems can be added to enlarge the number of processed satellites. The architecture is easily adaptable to handle multiple elements of the data stream in parallel, even when they are at different stages of processing.

The architecture of the Signal Processing Plane is heavily based on GNU Radio [32], a well-established framework that provides the signal processing runtime and processing blocks to implement software radio applications. Frameworks are a special case of software libraries – they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries: the overall program's flow of control is not dictated by the caller, but by the framework; and it can be extended by the user usually by selective overriding or specialized by user code providing specific functionality. Software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time.

GNU Radio's class hierarchy imposes a thread-per-block architecture that allows automatic scheduling in multicore processors, hiding all the complexity behind a simple and robust API. It uses shared memory to manage efficiently the flow of data between blocks, and offers a large set of well-programmed blocks that provide implementations for very common signal processing tasks. In contrast, GNU Radio does not provide any standard way to provide control over the blocks.

The user can build a receiver by creating a graph where the nodes are signal processing blocks and the lines represent the data flow between them. Conceptually, blocks process infinite streams of data flowing from their input ports to their output ports. The blocks' attributes include the number of input and output ports they have as well as the type of data that flows through each one of them. Once they are connected and form a *flowgraph*, the application can run and data will be put into the stream. As long as there is data, the working threads will run the code of the different blocks.

A key principle of reusable object-oriented design is: *Program to an interface, not an implementation* [33]. This principle is really about dependency relationships which have to be carefully managed in a complex application. It is easy to add a dependency on a class; however, the inverse is not that easy and getting rid of an unwanted dependency can turn into complicated refactoring work or even worse, blocking the user from reusing the code in another context. Isolating the interface from the implementation means the implementation can vary, and that is a healthy dependency relationship. This approach gives the developer flexibility, but it also separates the really valuable part, the design, from the implementation, which allows clients to be decoupled from the implementation. In fact, an abstract class gives you more flexibility when it comes to evolution. You can add new behavior without breaking clients.

One of the most attractive features of a software receiver is the possibility of interchanging algorithms (for instance, different implementations of signal acquisition and tracking) and observe its impact on the whole system, or establish fair comparisons among them. Moreover, these algorithms should be selectable at runtime. This kind of problem can be solved by means of the Strategy pattern [33], which defines a family of algorithms, encapsulates each one, and makes them interchangeable, letting the algorithms vary independently from clients that use them. The complexity in the instantiation of objects (for instance, concrete acquisition and tracking implementations) when only abstract interfaces are known can be addressed via the Factory Method pattern [33]. It consists in the encapsulation of the processes involved in the creation of objects, defining an interface for creating an object, but letting subclasses decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses. This approach eliminates the need to bind application-specific classes into the code, provides hooks for subclasses (thus making more flexible the creation of objects inside a class with a factory method than creating an object directly, for instance the addition of a new tracking method), and connects parallel class hierarchies (thus localizing knowledge of which classes belong together). A potential disadvantage of factory methods is that client applications might have to subclass the creator class just to create a particular concrete-product object.

In summary, we identified these requirements and features in terms of software architecture:

- We found C++ a convenient programming language for the implementation.
- The software receiver should address parallelization via multithreading.

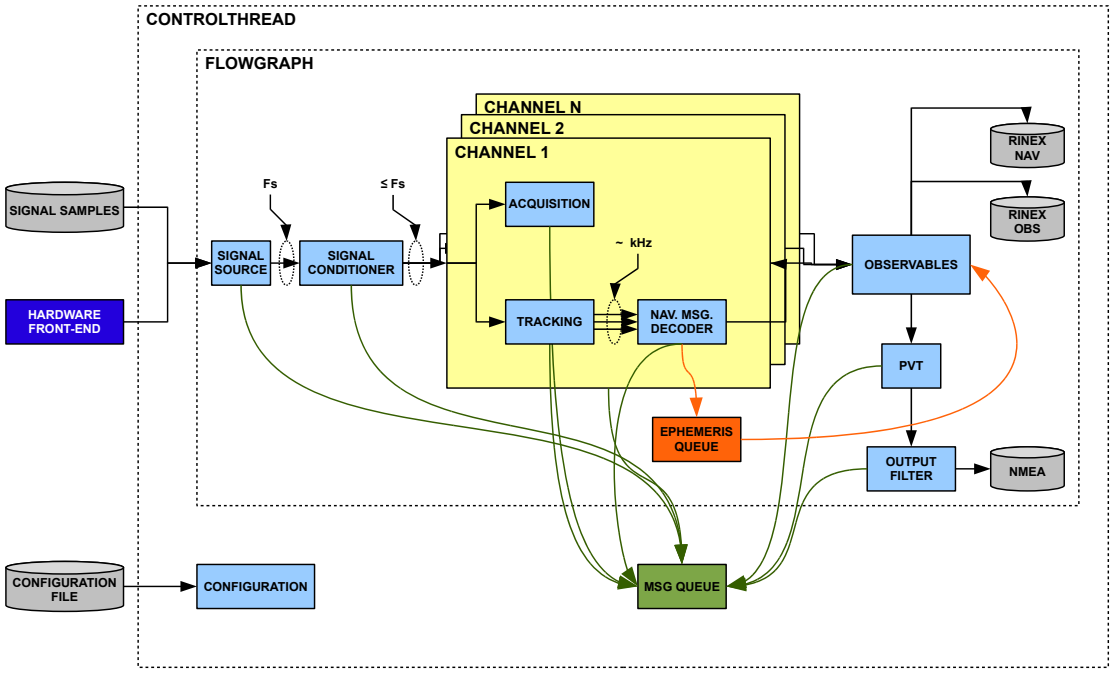

Fig. 1. Diagram of the modules that form the GNSS software receiver. Each module accepts multiple implementations, which can be selected by the user.

- It should perform reconfiguration during runtime.
- We need a thread-safe communication mechanism between processing modules.
- Decoupling between interfaces and implementations is crucial.
- We want to add new algorithms and to improve existing implementations without breaking anything.

2.3. Data sinks

Most geodetic processing software for GNSS data use a well-defined set of observables:

- the pseudorange (code) measurement, equivalent to the difference of the time of reception (expressed in the time frame of the receiver) and the time of transmission (expressed in the time frame of the satellite) of a distinct satellite signal,
- the carrier-phase measurement of one or more carriers (actually being a measurement on the beat frequency between the received carrier of the satellite signal and a receiver-generated reference frequency), and
- the observation time being the reading of the receiver clock at the instant of validition of the carrier-phase and/or the code measurements.

Receiver Independent Exchange Format (RINEX) is a data interchange format for raw satellite navigation system data, covering observables and the information contained in the navigation message broadcast by satellites. The most common version at present is 2.10, which enables storage of measurements from pseudorange, carrier-phase and Doppler systems for GPS or GLONASS, along with data from EGNOS and WAAS satellite based augmentation systems (SBAS), simultaneously. The need for improving the handling of the data files in case of files containing tracking data of more than one satellite system, each one with different observation types, led to significant modifications of the structure of data record. At this time of writing, the newest version is 3.01 [34], and includes defined formats for the modernized GPS signals, GLONASS, Galileo and SBAS. Hence,

observable and navigation RINEX files are one of the output formats of the proposed software receiver.

Commercial receivers usually do not provide access to such intermediate information but offer direct access to the position, velocity and time (PVT) solution and almanac data. The National Marine Electronics Association (NMEA) 0183 is the standard output format for commercial GPS receivers, and there is a plethora of compliant software available for handling and displaying such information. GNSS-SDR provides this output format for the PVT block module.

Finally, another interesting format to export geographic data is KML, directly importable by software applications such as Google Earth, Google Maps, and Google Maps for mobile. KML is an open standard officially named the OpenGIS KML Encoding Standard (OGC KML), and it is maintained by the Open Geospatial Consortium, Inc. (OGC).

In summary, we found useful the following formats for data output:

- RINEX.
- NMEA.
- KML.

3. IMPLEMENTATION DETAILS

GNSS-SDR's main method initializes the logging library, processes the command line flags, if any, provided by the user and instantiates a ControlThread[1] object. Its constructor reads the configuration file, creates a control queue and creates a flowgraph according to the configuration. Then, the program's main method calls the run() method of the instantiated object, an action that connects the flowgraph and starts running it. After that, and until a stop message is received, it reads control messages sent by the receiver's modules through a safe-thread queue and processes them. Finally, when a stop message is received, the main method executes the destructor of the ControlThread object, which deallocates memory, does other cleanup and gracefully exits the program.

As shown in Fig. 1, the software receiver is split into a Control Plane and a Signal Processing Plane. In the following subsections we provide details about their implementation.

3.1. CONTROL PLANE

The Control Plane is in charge of creating a flowgraph according to the configuration and then managing the modules. Configuration allows users to define in an easy way their own custom receiver by specifying the flowgraph (type of signal source, number of channels, algorithms to be used for each channel and each module, strategies for satellite selection, type of output format, etc.). Since it is difficult to foresee what future module implementations will be needed in terms of configuration, we used a very simple approach that can be extended without a major impact in the code. This can be achieved by simply mapping the names of the variables in the modules with the names of the parameters in the configuration.

Properties are passed around within the program using the ConfigurationInterface class. There are two implementations of this interface: FileConfiguration and InMemoryConfiguration. FileConfiguration reads the properties (pairs of property name and value) from a file and stores them internally. InMemoryConfiguration does not read from a file; it remains empty after instantiation and property values and names are set using the set_property method. FileConfiguration is intended to be used in the actual GNSS-SDR application whereas InMemoryConfiguration is intended to be used in tests to avoid file-dependency in the file system. Classes that need to read configuration parameters will receive instances of ConfigurationInterface from where they will fetch the values. For instance, parameters related to *SignalSource* should look like this:

```
SignalSource.parameter1=value1
SignalSource.parameter2=value2
```

The name of these parameters can be anything but one reserved word: implementation. This parameter indicates in its value the name of the class that has to be instantiated by the factory for that role. For instance, if we want to use the implementation *DirectResampler* for module *SignalConditioner*, the corresponding line in the configuration file would be

```
SignalConditioner.implementation=DirectResampler
```

Since the configuration is just a set of property names and values without any meaning or syntax, the system is very versatile and easily extendable. Adding new properties to the system only implies modifications in the classes that will make use of these properties. In addition, the configuration files are not checked against any strict syntax so it is always in a correct status (as long as it contains pairs of property names and values in INI format[2]).

Hence, the application defines a simple accessor class to fetch the configuration pairs of values and passes them to a factory class called GNSSBlockFactory. This factory decides,

[1] Actual name classes are denoted in sans–serif.

[2] An INI file is an 8-bit text file in which every property has a name and a value, in the form `name = value`. Properties are case-insensitive, and cannot contain spacing characters. Semicolons (`;`) indicate the start of a comment; everything between the semicolon and the end of the line is ignored.

according to the configuration, which class needs to be instantiated and which parameters should be passed to the constructor. Hence, the factory encapsulates the complexity of blocks' instantiation. With that approach, adding a new block that requires new parameters will be as simple as adding the block class and modifying the factory to be able to instantiate it. This loose coupling between the blocks' implementations and the syntax of the configuration enables extending the application capacities in a high degree. It also allows to produce fully customized receivers, for instance a testbed for acquisition algorithms, and to place observers at any point of the receiver chain.

The GNSSFlowgraph class is responsible for preparing the graph of blocks according to the configuration, running it, modifying it during run-time and stopping it. Blocks are identified by its role. This class knows which roles it has to instantiate and how to connect them to configure the generic graph that is shown in Fig. 1. It relies on the configuration to get the correct instances of the roles it needs and then it applies the connections between GNU Radio blocks to make the graph ready to be started. The complexity related to managing the blocks and the data stream is handled by GNU Radio's gr_top_block class. GNSSFlowgraph wraps the gr_top_block instance so we can take advantage of the GNSS block factory, the configuration system and the processing blocks. This class is also responsible for applying changes to the configuration of the flowgraph during run-time, dynamically reconfiguring channels: it selects the strategy for selecting satellites. This can range from a sequential search over all the satellites' ID to smarter approaches that determine what are the satellites most likely in-view based on rough estimations of the receiver position in order to avoid searching satellites in the other side of the Earth.

This class internally codifies actions to be taken on the graph. These actions are identified by simple integers. GNSSFlowgraph offers a method that receives an integer that codifies an action, and this method triggers the action represented by the integer. Actions can range from changing internal variables of blocks to modifying completely the constructed graph by adding/removing blocks. The number and complexity of actions is only constrained by the number of integers available to make the codification. This approach encapsulates the complexity of preparing a complete graph with all necessary blocks instantiated and connected. It also makes good use of the configuration system and of the GNSS block factory, which keeps the code clean and easy to understand. It also enables updating the set of actions to be performed to the graph quite easily.

The ControlThread class is responsible for instantiating the GNSSFlowgraph and passing the required configuration. Once the flowgraph is defined an its blocks connected, it starts to process the incoming data stream. The ControlThread object is then in charge of reading the control queue and processing all the messages sent by the the processing blocks via the thread-safe message queue.

3.2. SIGNAL PROCESSING PLANE

A key aspect of an object-oriented software design is the class hierarchy, depicted in Fig. 2. The notation is as follows: we used a very simplified version of the Unified Modeling Language (UML), a standardized general-purpose modeling language in the field of object-oriented software engineering. In this paper, classes are described as rectangles with two sections: the top section for the name of the class, and the bottom section for the methods of the class. A dashed arrow from *ClassA* to *ClassB* represents the dependency relationship. This relationship simply means that class A somehow depends upon class B. In C++ this almost always results in a #include. Inheritance models "is a" and "is like" relationships, enabling you to reuse existing data and code easily. When *ClassA* inherits from *ClassB*, we say that *ClassA* is the subclass of *ClassB* and *ClassB* is the superclass (or parent class) of *ClassA*. The UML modeling notation for inheritance is a line with a closed arrowhead pointing from the subclass to the superclass.

As shown in Fig. 2, gr_basic_block is the abstract base class for all signal processing blocks, a bare abstraction of an entity that has a name and a set of inputs and outputs. It is never instantiated directly; rather, this is the abstract parent class of both gr_hier_block2, which is a recursive container that adds or removes processing or hierarchical blocks to the internal graph, and gr_block, which is the abstract base class for all the processing blocks. A signal processing flow is constructed by creating a tree of hierarchical blocks, which at any level may also contain terminal nodes that actually implement signal processing functions.

Class gr_top_block is the top-level hierarchical block representing a flowgraph. It defines GNU Radio runtime functions used during the execution of the program: run(), start(), stop(), wait(), etc. As shown in Fig. 3, a subclass called GNSSBlockInterface is the common interface for all the GNSS-SDR modules. It defines pure virtual methods, that are required to be implemented by a derived class. Classes containing pure virtual methods are termed "abstract;" they cannot be instantiated directly, and a subclass of an abstract class can only be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class.

Subclassing GNSSBlockInterface, we defined interfaces for the receiver blocks defined in Fig. 1. This hierarchy,

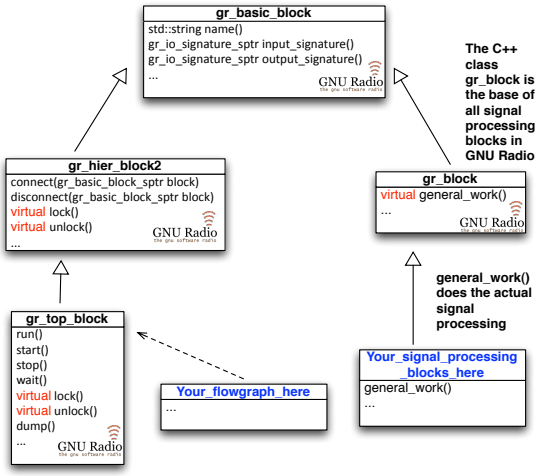

Fig. 2. Class hierarchy for the Signal Processing Plane.

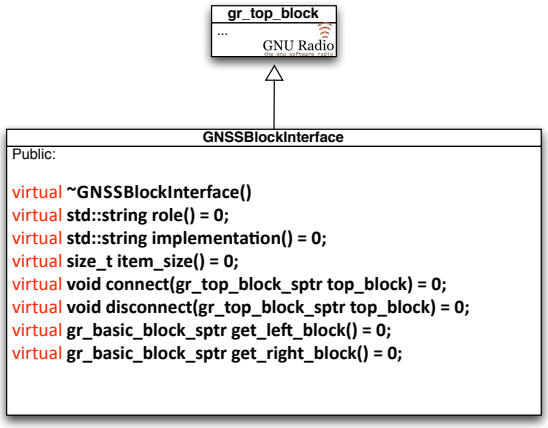

Fig. 3. General interface for signal processing blocks.

shown in Fig. 4, provides the definition of different algorithms and different implementations, which will be instantiated according to the configuration. This strategy allows multiple implementations sharing a common interface, achieving the objective of decoupling interfaces from implementations: it defines a family of algorithms, encapsulates each one, and makes them interchangeable. Hence, we let the algorithm vary independently from the program that uses it.

Hereafter, we describe GNSSBlockInterface subclasses. They are, again, abstract interfaces that defer instantiation to their own subclasses.

3.2.1. SIGNAL SOURCE

The Signal Source module is in charge of implementing the hardware driver, that is, the portion of the code that communicates with the RF front-end and receives the samples coming from the ADC. This communication is usually performed through USB or Ethernet buses. Since real-time processing requires a highly optimized implementation of the whole receiver, this module also allows to read samples from a file stored in a hard disk, and thus processing without time constraints. Relevant parameters of those samples are the intermediate frequency (or baseband I&Q components), the sampling rate and number of bits per sample, that must be specified by the user in the configuration file.

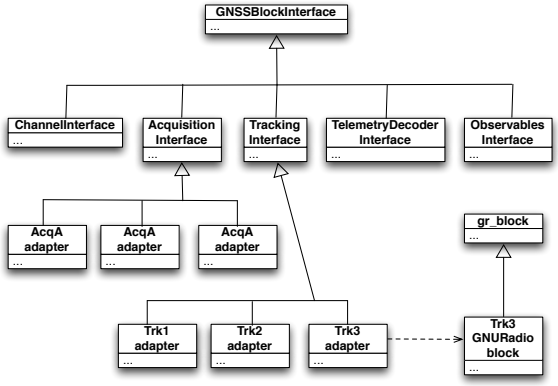

Fig. 4. Receiver's class hierarchy.

This module also performs bit-depth adaptation, since most of the existing RF front-ends provide samples quantized with 2 or 3 bits, while operations inside the processor are performed on 32- or 64-bit words, depending on its architecture. Although there are implementations of the most intensive computational processes (mainly correlation) that take advantage of specific data types and architectures for the sake of efficiency (such as the Single–Input Multiple–Data (SIMD)-based correlators presented in [6], exploiting a specific subset of assembly instructions for Intel processors that allows parallel computing), the approach is processor-specific and hardly portable. We suggest to keep signal samples in standard data types and letting the compiler select the best library version (implemented using SIMD or any other processor-specific technology) of the required routines for a given processor.

3.2.2. SIGNAL CONDITIONER

The signal conditioner is in charge of resampling the signal and delivering a reference sample rate to the downstream processing blocks, acting as a facade between the signal source and the synchronization channels, providing a simplified interface to the input signal. In case of multiband front-ends, this module would be in charge of providing a separated data stream for each band.

3.2.3. CHANNEL

A Channel encapsulates all signal processing devoted to a single satellite. Thus, it is a large composite object which encapsulates the acquisition, tracking and decoding modules. As a composite object, it can be treated as a single entity, meaning that it can be easily replicated. Since the number of channels is selectable by the user in the configuration file, this approach helps improving the scalability and maintainability of the receiver.

This module is also in charge of managing the interplay between acquisition and tracking. Acquisition can be initialized in several ways, depending on the prior information available (called cold start when the receiver has no information about its position nor the satellites almanac; warm start when a rough location and the approximate time of day are available, and the receiver has a recently recorded almanac broadcast; or hot start when the receiver was tracking a satellite and the signal line of sight broke for a short period of time, but the ephemeris and almanac data is still valid, or this information is provided by other means), and an acquisition process can finish deciding that the satellite is not present, that longer integration is needed in order to confirm the presence of the satellite, or declaring the satellite present. In the latter case, acquisition process should stop and trigger the tracking module with coarse estimations of the synchronization parameters. The mathematical abstraction used to design this logic is known as finite state machine (FSM), that is a behavior model composed of a finite number of states, transitions between those states, and actions. For the implementation, we used the Boost.Statechart library [35], which provides desirable features such as support for asynchronous state machines, multi-threading, type-safety, error handling and compile-time validation.

At the time of writing, we provide an implementation that works with the GPS L1 C/A signal. More complex channel implementations could accommodate more sophisticated strategies such as vector tracking loops involving signals from more than one satellite and optimizations performed directly in the position domain [36].

3.2.4. ACQUISITION

The first task of a GNSS receiver is to detect the presence or absence of in-view satellites. This is done by the acquisition system process, which also provides a coarse estimation of two signal parameters: the frequency shift f_d with respect to the nominal IF frequency, and a delay term τ which allows

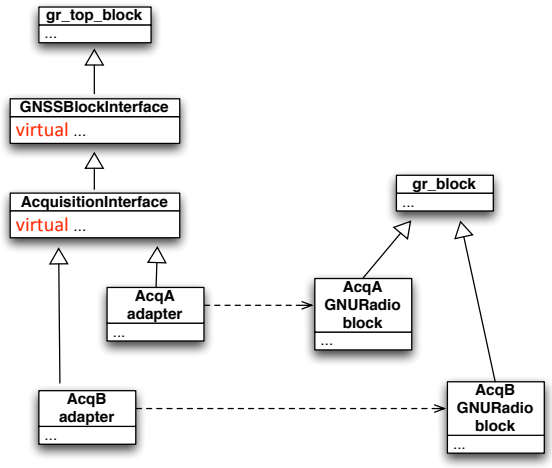

Fig. 5. Class hierarchy for the Acquisition module. AcqA and AcqB adapter classes wrap your custom acquisition block to the common interface expected by AcquisitionInterface. AcqA and AcqB GNU Radio blocks are different implementations of acquisition. Actual signal processing is done here. You can use existing GNU Radio blocks or implement a new one by yourself.

the receiver to create a local code aligned with the incoming code.

AcquisitionInterface is the common interface for all the acquisition algorithms and their corresponding implementations. Algorithms' interface, that may vary depending on the use of information external to the receiver, such as in Assisted GNSS, is defined in classes referred to as *adapters* (see Fig. 5). These adapters wrap the GNU Radio blocks interface into a compatible interface expected by AcquisitionInterface. This allows the use of existing GNU Radio blocks derived from gr_block, and ensures that newly developed implementations will also be reusable in other GNU Radio-based applications. Moreover, it adds still another layer of abstraction, since each given acquisition algorithm can have different implementations (for instance using different numerical libraries). In such a way, implementations can be continuously improved without having any impact neither on the algorithm interface nor the general acquisition interface.

Software-based acquisition methods can consist of a serial search, a parallel frequency search based on the Fast Fourier Transform [13] (that can be implemented with specialized and highly optimized libraries such as FFTW [37]), using coherent or non-coherent integration [12], or even taking advantage of signals from different frequency bands [38].

3.2.5. TRACKING

When a satellite is declared present, the parameters estimated by the acquisition module are then fed to the receiver tracking module, which represents the second stage of the signal processing unit, aiming to perform a local search for accurate estimates of code delay and carrier phase, and following their eventual variations. Possible algorithms for code delay tracking include the Delay Lock Loop (DLL) and its multiple variants, along with code lock detectors that decide whether the tracking loop is effectively locked to the code phase of the received signal or a loss-of-lock has occurred, while Phase Lock Loops (PLL) and Frequency Lock Loops (FLL) are the usual approaches for carrier tracking (also encompassing different carrier phase loop discriminators and carrier lock detectors [39]). Since the time spent in the transition from acquisition to tracking is unknown, a counter is implemented in order to estimate on which sample the spreading code sequence begins after a certain period of time.

Again, a class hierarchy consisting of a TrackingInterface class and subclasses implementing algorithms provides a way of testing different approaches, with full access to their parameters. In additional to classical tracking loops, this architectures leaves room for other approaches such as the block adjustment of synchronizing signal (BASS) method [12], strategies based on combinations of signals in different bands [40] or ultra-tight hybridization schemes with inertial measurement units [41].

3.2.6. DECODING OF NAVIGATION MESSAGE

Most of GNSS signal links are modulated by a navigation message containing the time the message was transmitted, orbital parameters of satellites (also known as ephemeris) and an almanac (information about the general system health, rough orbits of all satellites in the network as well as data related to error correction). Navigation data bits are structured in words, pages, subframes, frames and superframes. Sometimes, bits corresponding to a single parameter are spread over different words, and values extracted from different frames are required for proper decoding. Some words are for synchronization purposes, others for error control an others contain actual information. There are also error control mechanisms, from parity checks to forward error correction (FEC) encoding and interleaving, depending on the system. All this decoding complexity is managed by a finite state machine implemented with the Boost.Statechart library [35]. Details on the structure of the navigation message are reported in the related interface specification documents: see [42] for GPS L1 C/A and L2C, [43] for GPS L1C, [44] for GPS L5, [45] for GLONASS, [46] for Galileo and [47] for SBAS. Class hierarchy for these implementations mimics the architecture already presented for acquisition and tracking: a TelemetryDecoderInterface provides a single interface for decoding the navigation message of different systems.

3.2.7. OBSERVABLES

GNSS systems provide different kinds of observations. The most commonly used are the code observations, also called pseudoranges. The *pseudo* comes from the fact that on the receiver side the clock error is unknown and thus the measurement is not a pure range observation. High accuracy applications also use the carrier phase observations, which are based on measuring the difference between the carrier phase transmitted by the GNSS satellites and the phase of the carrier generated in the receiver. Both observables are computed from the outputs of the tracking module and the decoding of the navigation message.

This module collects all the data provided by every tracked channel, aligns all received data into a coherent set, and computes the observables. Mathematical details can be found in [48]. The output format for this data is in form of RINEX file, as described in Section 2.3. RINEX files pave the way to high accuracy positioning, since they can be directly used by applications and libraries that implement the corresponding algorithms.

3.2.8. PVT

Although data processing for obtaining high–accuracy PVT solutions is out of the scope of GNSS-SDR, we provide a module that can compute a simple least square solution and leaves room for more sophisticated positioning methods. The integration with libraries and software tools that are able to deal with multi-constellation data such as GPSTk or gLAB [49] appears as a viable solution for high performance, completely customizable GNSS receivers.

3.3. OTHER FEATURES

- **Development ecosystem.** Infrastructure for project management, code development and efficient communication among users and developers is a key aspect in software projects. The website should be well designed in terms of usability, functionality and extendability, ensuring an enjoyable and appealing user experience. We used Drupal [50] as a content management system. The result can be found at `www.gnss-sdr.org`. For the source code, an application that automates the process of keeping an annotated history of the project, allowing reversion of code changes, change tracking, and bug tracking is essential. We used the service provided by SourceForge, which allows access to the code using two major revision control systems such as Subversion [51] and git [52], along with a distribution list that facilitates communication among users.

- **Documentation.** This is of paramount importance for users, developers, testers, software architects and students. We used Doxygen [53], a tool for writing software reference documentation. The documentation is written within the C++ code, in form of comments, and is thus relatively easy to generate (since it can be written along with the source code) and keep up to date. Doxygen scans the code, extracts the documentation and dumps it in HTML, LaTeX, RTF or XML formats, cross-referencing documentation and code, so that the reader of a document can easily refer to the actual code. It also automatically generates dependency graphs, inheritance diagrams, and collaboration diagrams. In addition to code documentation, the project web page provides with detailed instructions about the installation, usage, coding style and general information about the program.

- **Building tool.** The build process should be easily maintained and highly portable. When not thought out well, development time shifts towards build system tweaking instead of source file coding. We used Boost.Build [54], a tool that takes care about compiling the sources with the right options, creating static and shared libraries, making executables, using a variety of compilers on the most common operating systems. The developer is only required to provide a simple and high level building description (naming executables and libraries and listing their sources) and Boost.Build provides many built-in features that can be combined to produce arbitrary building configurations, such as debug and release variants or single- and multi-threaded builds, automatically translating the requested properties into appropriate command-line flags for invoking toolset components like compilers and linkers. Regarding the compiler, we used the GNU Compiler Collection [55].

- **Testing.** We suggest a test–driven development approach. This methodology is claimed to offer valuable benefits to software development: it facilitates change, simplifies integration, automatizes documentation, helps separate the interface from the implementation, increases developers productivity, and plays a central role in the software quality assurance process. For unit testing, we found the Google C++ Testing Framework (gtest) [56] useful and lightweight.

- **Logging.** Logging is important for debugging and tracing purposes. The developer might also want to run some interesting statistics on logs, and study usage patterns. In GNSS-SDR, logging is handled by Google Logging Library (google-glog) [57], a library that implements application-level logging. It provides simple yet powerful APIs to various log events in the program. Messages can be logged by severity level, and the users can control logging behavior from the command line, log based on conditionals, abort the program with stack trace when expected conditions are not met, and introduce their own verbose logging levels.

- **Profiling.** We used Google Performance Tools (google–perftools) [58], a collection of a high–performance multi–threaded memory allocators implementation, plus some performance analysis tools that allow to automatize the profiling process, identify computational bottlenecks, and help the developers to focus their optimization efforts.

4. CONCLUSIONS

This paper presented an open source GNSS software defined receiver, discussing aspects about its design and implementation. The proposed software receiver targets multi-constellation/multi-frequency architectures, pursuing the goals of efficiency, modularity, interoperability, and flexibility demanded by user domains that require non-standard features. The source code was released under the GNU General Public License (GPL), thus ensuring the freedom of modifying, sharing, and using the code for any purpose. This secures practical usability, inspection, and continuous improvement by the research community, allowing the discussion based on tangible code and the analysis of results obtained with real signals. Hence, it is also intended to be a framework for algorithm testing and an educational tool. The website available at `www.gnss-sdr.org` provides information and detailed instructions, as well as a link to the source code.

Although GNSS-SDR is still far from full-featured commercial software receivers, it constitutes a free platform that can be continuously improved by peer-reviewing and contributions from users and developers around the world, unleashing the potential of collaborative research in the field of GNSS software receivers.

5. ACKNOWLEDGEMENTS

Authors want to thank the whole open source community, and specifically to other pioneering open source projects that have inspired this work: C. Kelley's OpenSource GPS [59, 60], a software for x86 PCs that allows acquisition, tracking and demodulation of signals from GPS satellites (requiring a Zarlink's GP2021 GPS correlator chip) and G. Heckler's gps-sdr [61], which implemented a x86 Linux-based GPS software receiver, including correlators using SIMD technology [6]. This work has been partially supported by the Spanish Science and Technology Commission under Project TEC2008-02685/TEC (NARRA) and by the European Com-

mission in the framework of the COST Action IC0803 (RFC-SET).

REFERENCES

[1] C. Fernández-Prades, L. Lo Presti, and E. Falleti, "Satellite radiolocalization from GPS to GNSS and beyond: Novel technologies and applications for civil mass-market," *Proceedings of the IEEE*, Nov. 2011, In Press. DOI: 10.1109/JPROC.2011.2158032.

[2] D. Akos, *A Software Radio Approach to Global Navigation Satellite System Receiver Design*, Ph.D. thesis, College of Engineering and Technology, Ohio University, Aug. 1997.

[3] K. Krumvieda, P. Madhani, C. Cloman, E. Olson, J. Thomas, P. Axelrad, and W. Kober, "A complete IF software GPS receiver: A tutorial about the details," in *Proc. of the 14th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GPS'01)*, Salt Lake City, UT, Sept 2001, pp. 789–829.

[4] V. Chakravarthy, J. Tsui, D. Lin, and J. Schamus, "Software GPS receiver," *GPS Solutions*, vol. 5, no. 2, pp. 63–70, Oct. 2001.

[5] B. M. Ledvina, S. P. Powell, P. M. Kintner, and M. L. Psiaki, "A 12–channel real–time GPS L1 software receiver," in *Proc. of the National Technical Meeting of the Institute of Navigation (ION NTM'03)*, Anaheim, CA, Jan. 2003, pp. 767–782.

[6] G. W. Heckler and J. L. Garrison, "SIMD correlator library for GNSS software receivers," *GPS Solutions*, vol. 10, no. 4, pp. 269–276, Nov. 2006.

[7] H. Hurskainen, J. Raasakka, T. Ahonen, and J. Nurmi, "Multicore software–defined radio architecture for GNSS receiver signal processing," *EURASIP Journal on Embedded Systems*, vol. 2009, 2009, Article ID 543720.

[8] T. E. Humphreys, J. A. Bhatti, T. Pany, B. M. Ledvina, and B. W. O'Hanlon, "Exploiting multicore technology in software–defined GNSS receivers," in *Proc. of the 22nd International Meeting of the Satellite Division of The Institute of Navigation (ION GNSS'09)*, Savannah, GA, Sept. 2009, pp. 326–338.

[9] A. Mitelman, J. Almqvist, R. Håkanson, D. Karlsson, F. Lindström, T. Renström, C. Ståhlberg, and J. Tidd, "Testing software receivers," *GPS World*, vol. 20, no. 12, pp. 28–34, Dec. 2009.

[10] T. Hobiger, T. Gotoh, J. Amagai, Y. Koyama, and T. Kondo, "A GPU based real–time GPS software receiver," *GPS Solutions*, vol. 14, no. 2, pp. 207–216, Mar. 2010.

[11] X. Li and D. Akos, "Implementation and performance of clock steering in a software GPS L1 single frequency receiver," *Navigation: Journal of The Institute of Navigation*, vol. 57, no. 1, pp. 69–85, Spring 2010.

[12] J. Bao-Yen Tsui, *Fundamentals of Global Positioning System Receivers. A Software Approach*, John Wiley & Sons, Inc., Hoboken, NJ, 2nd edition, 2005.

[13] K. Borre, D. M. Akos, N. Bertelsen, P. Rinder, and S. H. Jensen, *A Software–Defined GPS and Galileo Receiver. A Single–Frequency Approach*, Applied and Numerical Harmonic Analysis. Birkhäuser, Boston, MA, 2007.

[14] M. G. Petovello, C. O'Driscoll, G. Lachapelle, D. Borio, and H. Murtaza, "Architecture and benefits of an advanced GNSS software receiver," *Positioning*, vol. 1, no. 1, pp. 66–78, 2009.

[15] F. Principe, G. Bacci, F. Giannetti, and M. Luise, "Software-defined radio technologies for GNSS receivers: A tutorial approach to a simple design and implementation," *International Journal of Navigation and Observation*, vol. 2011, pp. 1–27, 2011, Article ID 979815, DOI:10.1155/2011/979815.

[16] C. Fernández–Prades, C. Avilés, L. Esteve, J. Arribas, and P. Closas, "Design patterns for GNSS software receivers," in *Proc. of the 5th ESA Workshop on Satellite Navigation Technologies (NAVITEC'2010)*, ESTEC, Noordwijk, The Netherlands, Dec. 2010, DOI:10.1109/NAVITEC.2010.5707981.

[17] M. Anghileri, T. Pany, D. Sanromà-Güixens, J.-H. Won, A. Sicramaz-Ayaz, C. Stöber, I. Krämer, D. Dötterböck, G. W. Hein, and B. Eissfeller, "Performance evaluation of a multi–frequency GPS/Galileo/SBAS software receiver," in *Proc. of the 20th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS'07)*, Fort Worth, TX, Sept. 2007, pp. 2749–2761.

[18] C. Stöber, M. Anghileri, A. Sicramaz Ayaz, D. Dötterböck, I. Krämer, V. Kroppand J.-H. Won, B. Eissfeller, D. Sanromà Güixens, and T. Pany, "ipexSR: A real-time multi-frequency software GNSS receiver," in *Proc. of the 52nd International Symposium ELMAR-2010*, Zadar, Croatia, Sept. 2010, pp. 407–416.

[19] M. Fantino, A. Molino, and M. Nicola, "N–Gene GNSS receiver: Benefits of software radio in navigation," in *Proc. of the European Navigation Conference - Global Navigation Satellite Systems (ENC-GNSS)*, Naples, Italy, May 2009.

[20] B. W. Tolman, R. B. Harris, T. Gaussiran, D. Munton, J. Little, R. Mach, S. Nelsen, B. Renfro, and D. Schlossberg, "The GPS toolkit - open source GPS software," in *Proc. of the 17th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS'04)*, Long Beach, CA, 21–24 Sept. 2004, pp. 2044–2053.

[21] D. Salazar, M. Hernández-Pajares, J. M. Juan, and J. Sanz, "GNSS data management and processing with the GPSTk," *GPS Solutions*, vol. 14, no. 3, pp. 293–299, June 2010.

[22] Ettus Research, "USRP motherboard datasheet. Universal Software Radio Peripheral. The foundation for complete software radio systems," http://www.ettus.com/downloads/ettus_ds_usrp_v7.pdf, Retrieved: October 21, 2010.

[23] Ettus Research, "TX and RX daughterboards for the USRP^{TM} software radio system," http://www.ettus.com/downloads/ettus_daughterboards.pdf, Retrieved: August 29, 2011.

[24] SiGe Semiconductor, "SE4120L GNSS receiver IC datasheet," May 26, 2009, Ottawa, ON, Canada.

[25] Maxim Integrated Products, Sunnyvale, CA, *MAX2769 Universal GPS Receiver*, June 2010, Ref. 19-0791; Rev 2.

[26] N. Falk, T. Hartmann, H. Kern, B. Riedl, T. Pany, R. Wolf, and J. Winkel, "SX-NSR 2.0 A multi-frequency and multi-sensor software receiver with a quad-band RF front end," in *Proc. of the 23rd International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS 2010)*, Portland, OR, Sept. 2010, pp. 1395–1401.

[27] G. Rivela, P. Scavini, D. Grasso, M. Castro, A. Calcagno, G. Avellone, A. Di Mauro, G. Cali, and S. Scaccianoce, "A low power RF front-end for L1/E1 GPS/Galileo and GLONASS signals in CMOS 65nm technology," in *Proc. of International Conference on Localization and GNSS (ICL-GNSS)*, Tampere, Findland, June 2011, pp. 7–12.

[28] M. Detratti, E. López, E. Pérez, and R. Palacio, "Dual-frequency RF front end solution for hybrid Galileo/GPS mass market receiver," in *Proc. of the IEEE Consumer Communications and Networking Conference (CCNC '08)*, Las Vegas, NV, Jan. 2008, pp. 603–607.

[29] J. Wu, P. Jiang, D. Chen, and J. Zhou, "A dual-band GNSS RF front end with a pseudo-differential LNA," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 58, no. 3, pp. 134–138, March 2011.

[30] *ISO/IEC 14882:2011(E) Programming Languages – C++, Third Edition*, 2011.

[31] B. P. Douglass, *Real–Time Design Patterns: Robust Scalable Architecture for Real–Time Systems*, Addison Wesley, Upper Saddle River, NJ, 2002.

[32] "GNU Radio," http://gnuradio.org, Retrieved: August 29, 2011.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object–Oriented Software*, Addison Wesley, Upper Saddle River, NJ, 1995.

[34] W. Gurtner and L. Estey, *RINEX. The Receiver Independent Exchange Format Version 3.01*, June 2009.

[35] "The Boost Statechart Library," www.boost.org/doc/libs/release/libs/statechart/doc/index.html, Retrieved: August 29, 2011.

[36] P. Closas and C. Fernández-Prades, "Bayesian nonlinear filters for Direct Position Estimation," in *Proc. of IEEE Aerospace Conference*, Big Sky, MT, March 2010, pp. 1–12, DOI: 10.1109/AERO.2010.5446676.

[37] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, Special issue on "Program Generation, Optimization, and Platform Adaptation".

[38] C. Gernot, K. O'Keefe, and G. Lachapelle, "Assessing three new GPS combined L1/L2C acquisition methods," *IEEE Transactions of Aerospace and Electronic Systems*, vol. 3, no. 47, pp. 2239–2247, July 2011, DOI: 10.1109/TAES.2011.5937297.

[39] E. D. Kaplan and C. J. Hegarty, Eds., *Understanding GPS. Principles and Applications*, Artech House, Norwood, MA, 2nd edition, 2006.

[40] D. Megahed, C. O'Driscoll, and G. Lachapelle, "Performance evaluation of combined L1/L5 Kalman filter-based tracking versus standalone L1/L5 tracking in challenging environments," in *Proc. of the 22nd International Meeting of the Satellite Division of The Institute of Navigation (ION GNSS)*, Savannah, GA, Sept. 2009, pp. 2591–2601.

[41] C. Fernández-Prades, P. Closas, and J. Vilà-Valls, "Nonlinear filtering for ultra-tight GNSS/INS integration," in *Proc. of the IEEE International Conference on Communications (ICC 2010)*, Cape Town (South Africa), May 2010, pp. 1–5, DOI: 10.1109/ICC.2010.5502037.

[42] Science Applications International Corporation, *Interface Specification IS-GPS-200 Revision E. Navstar GPS Space Segment/Navigation User Interfaces*, El Segundo, CA, June 8, 2010.

[43] Science Applications International Corporation, *Interface Specification IS-GPS-800 Revision A. Navstar GPS Space Segment/User Segment L1C Interfaces*, El Segundo, CA, June 8, 2010.

[44] Science Applications International Corporation, *Interface Specification IS-GPS-705 Revision A. Navstar GPS Space Segment/User Segment L5 Interfaces*, El Segundo, CA, June 8, 2010.

[45] Russian Institute of Space Device Engineering, Moscow, Russia, *Global Navigation Satellite System GLONASS. Interface Control Document. Navigational radiosignal in bands L1, L2*, 2008, Version 5.1 downloadable from http://www.glonass-ianc.rsa.ru.

[46] European Union, *European GNSS (Galileo) Open Service. Signal In Space Interface Control Document. Ref: OS SIS ICD, Issue 1*, February 2010.

[47] RTCA, Washington, DC, *Minimum Operational Performance Standards for Global Positioning System/Wide Area Augmentation System Airborne Equipment, DO–229D*, Dec. 13 2006.

[48] A. Leick, *GPS Satellite Surveying*, John Wiley & Sons, Inc., Hoboken, NJ, 3rd edition, 2004.

[49] M. Hernández-Pajares, J. M. Juan, J. Sanz, P. Ramos-Bosch, A. Rovira-García, D. Salazar, J. Ventura-Traveset, C. López-Echazarreta, and G. Hein, "The ESA/UPC GNSS-Lab Tool (gLAB)," in *Proc. of the 5th ESA Workshop on Satellite Navigation Technologies (NAVITEC'2010)*, ESTEC, Noordwijk, The Netherlands, Dec. 2010, DOI: 10.1109/NAVITEC.2010.5708032.

[50] "Drupal," http://drupal.org, Retrieved: August 29, 2011.

[51] "Apache Subversion," http://subversion.apache.org, Retrieved: August 29, 2011.

[52] "git. The fast version control system," http://git-scm.com, Retrieved: August 29, 2011.

[53] "Doxygen," http://www.stack.nl/ dimitri/doxygen/, Retrieved: August 29, 2011.

[54] "Boost.build v2," http://www.boost.org/boost-build2/doc/html/index.html, Retrieved: August 29, 2011.

[55] "GCC, the GNU Compiler Collection," http://gcc.gnu.org, Retrieved: August 29, 2011.

[56] "Google C++ Testing Framework (googletest)," http://code.google.com/p/googletest/, Retrieved: August 29, 2011.

[57] "Google logging library (google-glog). Logging library for C++," http://code.google.com/p/google-glog/, Retrieved: August 29, 2011.

[58] "Google Performance Tools (google-perftools)," http://code.google.com/p/google-perftools/, Retrieved: August 29, 2011.

[59] C. Kelley, J. Barnes, and J Cheng, "OpenSource GPS. Open source software for learning about GPS," in *Proc. of the 15th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GPS'02)*, Portland, OR, Sept. 2002.

[60] "OpenSourceGPS," http://sourceforge.net/projects/osgps/, Retrieved: August 29, 2011.

[61] "GPS-SDR," https://github.com/gps-sdr, Retrieved: August 29, 2011.