Highly Efficient Representation of Reconfigurable Code Based on a Radio Virtual Machine: Optimization to Any Target Platform

Vladimir Ivanov[1], Yong Jin[2], Seungwon Choi[2], Guiseppe Destino[3], Markus Mueck[4], Valerio Frascolla[4]
[1]State University of Aerospace Instrumentation, St. Petersburg, Russia
[2]Hanyang University, Korea
[3]University of Oulu, Finland
[4]INTEL, Germany

***Abstract*— ETSI has developed a novel Software Radio Reconfiguration framework encompassing technical, certification and security solutions. Compared to legacy Software Reconfiguration technology, such as the Software Communications Architecture, the ETSI solution is designed for lowest overall power consumption and efficiency. For this purpose, a novel approach for Code Portability has been developed – a Radio Virtual Machine based mechanism allows converting a given algorithm into a generic representation, which is then, optimized for the specific hardware resources available on a target platform. This contribution explains the basic principles and outlines how Code Portability is achieved while meeting the objectives in terms of power consumption and complexity.**

Keywords*— *Software Reconfiguration, Software Defined Radio, Radio Virtual Machine

I. INTRODUCTION

ETSI has recently published a Software Reconfiguration approach comprising an entire ecosystem including technical, regulation and security solutions [1][2][3][4][5][6] for altering Radio parameters of a Mobile Device through provision of software components. This activity coincides with an effort of the European Commission to revise the corresponding European Radio Directive. The previously existing R&TTE Directive [10] is replaced by the novel Radio Equipment Directive [9] whose articles 3(3)(i) and 4 provide specific provisions for Software Reconfiguration. ETSI has discussed technical solutions for addressing the new regulation framework documented in [11].

In this context, a key question relates to Code Portability and how it can be achieved while maintaining high power efficiency and low overall complexity. Alternative approaches, such as the Software Communications Architecture [12][13], introduce an isolation of the software from specific radio hardware or implementations by standardized APIs which leads to highly portable solutions at the cost of introducing an additional layer accessing hardware resources through APIs. ETSI has chosen a different trade-off based on a so-called Radio Virtual Machine (RVM) approach [6]. In the sequel of this paper, we outline how this approach will provide Code portability while ensuring efficiency in terms of overall power consumption and complexity.

The sequel of the paper is organized as follows. Section II introduces the overall ETSI Software Reconfiguration framework and comments on differences with respect to alternative solutions. Section III introduces the Radio Virtual Machine concept, followed by a discussion on Code Portability in section IV. Section V finally gives a conclusion.

II. THE ETSI RRS SOFTWARE RADIO RECONFIGURATION FRAMEWORK AND COMPARISON TO ALTERNATIVE APPROACHES

A. ETSI RRS Approach

The overall ETSI Software Radio Reconfiguration eco-system and the Mobile Device reconfiguration architecture are outlined in Fig. 1. It provides a complete framework covering technical, regulation and security solutions for software reconfiguration. As outlined in further detail in [2], Fig. 1 shows the reconfigurable mobile device architectural components related to the radio reconfiguration as well as the related entities. There were identified four interfaces to handle radio reconfigurations: Multiradio Interface (MURI), Unified Radio Application Interface (URAI), Reconfigurable Radio Frequency Interface (RRFI) and Radio Programming Interface (RPI).

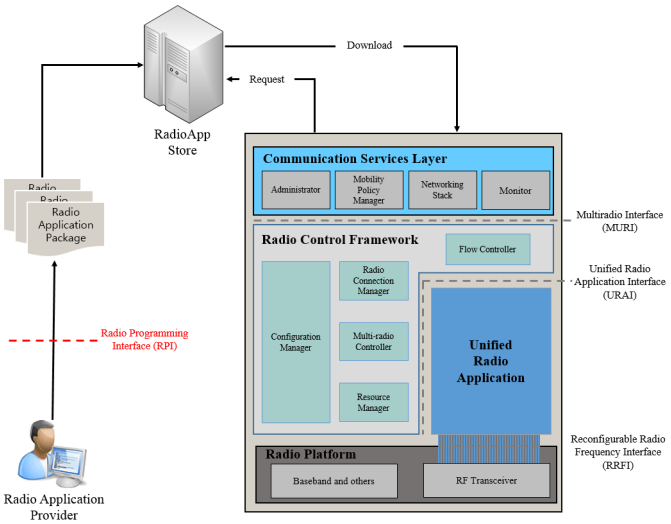

Fig. 1. The Reconfigurable Mobile Device Architecture.

As shown in the figure, the following components can be identified:

- Communication Services Layer (CSL): This layer can be interpreted as the "Mobile Device Manufacturer" domain, which allows to control the (un)installation of new Radio Applications (RAs), the definition of policies on how to select suitable Radio Access Technologies (RATs), to access to lower layer configuration information through the monitor and to access the dataflow. Four logical entities are defined: Administration, Mobility Policy Manager, Networking Stack and Monitor.
- Radio Control Framework (RCF): The RCF takes commands from the CSL and translates those to specific instructions to the underlying hardware resources, e.g. for Radio Applications (un)installation, execution, etc. Five logical entities are defined: Configuration Manager, Radio Connection Manager, Multi-Radio Controller, Resource Manager and Flow Controller.
- Unified Radio Applications (URA): Radio Applications are the software packages to be executed on the underlying hardware. Since all RAs exhibit a common behavior from the reconfigurable mobile device perspective, those Radio Applications are called URAs.
- Radio Platform (consisting of RF Transceiver, Baseband, etc.): The Radio Platform consists of computational and radio elements.
- The representation of generic code to be used for the Radio Programming Interface is defined in [6].

It should be noted that the ETSI Software Radio Reconfiguration must not necessarily used entirely for all applications. For example, the usage of the Multiradio Interface between CSL and RCF [3] alone would enable a manufacturer to use all basic reconfiguration functions such as (un)installation of RadioApps, execution of RadioApps and etc. Albeit portability will not be guaranteed without using the Radio Programming Interface framework. However, it may not be required in the first generation of a reconfigurable platform for example.

B. Alternative Approaches

A multitude of software reconfiguration approaches exist. Among those, the Software Communication Architecture (SCA) [12][13] is a very prominent solution. Therefore, a short overview is given and the differences to the ETSI approach are outlined.

As it is mentioned in [12], *the Software Communications Architecture (SCA) is published by the Joint Tactical Networking Center (JTNC). This architecture was developed to assist in the development of Software Defined Radio (SDR) communication systems, capturing the benefits of recent technology advances, which are expected to greatly enhance interoperability of communication systems and reduce development and deployment costs*. As furthermore stated in [13], *a fundamental feature of the SCA is the separation of waveforms from the radio's operating environment. Waveform portability is enhanced by establishing a standardized host environment for waveforms, regardless of other radio characteristics … The waveform software is isolated from specific radio hardware or implementations by standardized APIs.*

Nevertheless, the portability problem was not resolved within SCA, see for example, [14], [18] where it was pointed out as a primary problem of SCA. SCA (any version) middleware separates and isolates software from hardware and therefore does not allow a joint optimization of hardware and software which is the main source of efficiency for embedded devices. Although the middleware of SCA is quite sophisticated, it is too redundant and, thus, not efficient enough for commercial applications. The development of such middleware is quite costly for civil industry. Historically, SCA was designed based on the distributed computing approach, but the modern terminals are built based on System-on-Chips (SoC) where multiple Intellectual Property (IP) cores are integrated into a single chip. Still, since SoC-based technology does not assume distributed internal communications, it is not reasonable to support the baseline "client-server" model and sophisticated hardware agnostic transactions among software components. Meanwhile, since the "client-server" model proposed by the Object Management Group (OMG) is not formal; thus, it cannot support a formal verification, which is critical due to the software complexity for emerging Multi-RAT (multiple Radio Access Technology) Mobile Devices working in a heterogeneous wireless network environment.

An alternative approach of a Radio Virtual Machine (RVM) as the Java Virtual Machine (JVM) was considered in [15], [16]. The implementation aspects of the RVM were studied in [16], [17]. However, none of these sources treated the problem of an RVM architecture efficiency. The authors rather focused on existing VMs based on the von Neumann architecture and evaluated corresponding overheads.

III. Radio Virtual Machine

The ETSI Software Reconfiguration solution treats wireless communications as **Radio Computing,** where they can be considered as a domain-specific embedded computing which deals with requirements and algorithms specific for wireless communications and related to radio waves processing. It is assumed that the radio computer interacts with the external world (radio spectrum) with a capability of receiving and emitting radio waves.

In this paper, we provide a contribution for the development of a specific concurrent Model of Computations (MoC) and implementation of the MoC as a Radio Virtual Machine supporting the following features, which define its efficiency:

- True concurrency
- Combining message passing and shared memory architecture in arbitrary proportion;
- Vertical and horizontal parallelism;
- Minimal control overhead;

- Aggregation of RVM into more complex RVM;
- Adjusted set of operations.

A. General Concept

We introduce a set of abstract resources consisting of Abstract Processing Elements (APE) and Data Objects. APEs abstract computational elements executing any operation from the initially given set of operations, which are typical for radio processing. Such a set is called *Radio Library*. Data Objects abstract the memory notion. The Radio Library provides a set of operations, which can be carried out on data.

The specific RVM plays a role of the Turing machine in applications related to radio processing. The specific RVM is created for a given algorithm according to the following procedure. The RVM allocates corresponding resources such as APEs and Data Objects for each operator and for data from the algorithm. The Radio Library provides operators semantics. Only one APE is allocated for each operator and only one Data Object is allocated for each data element. The specific RVM begins to work immediately after resource allocation and data initialization. All APEs work asynchronously and concurrently. An individual APE executes the allocated operator if all input Data Objects are full. APEs access Data Objects with operations "read", "read-erase" (re), "read-erase-write" (rew) or "write" (w). After reading input data from Data Objects, the APE executes the allocated operator and, if output Data Objects are empty, then the APE writes processed data. Any full output Data Object blocks the corresponding writing operation. Full data objects and executing APEs are considered as active. APEs are treated as inactive after finishing operator executions and writing output data. Inactive APEs are returned to the APE's pool. Empty Data Objects, which are not connected with APEs, are also returned to the Data Object's pool

B. RVM Architecture

The Radio Virtual Machine (RVM) is a **universal** abstract machine, which is capable to execute concurrent computations expressed by **specific** RVMs. The RVM architecture is shown in Fig. 2. The block "Basic Operations" downloads operations from the Radio Library. Operations are defined by operator identifiers. The RVM data path consists of Data Objects (DO), Abstract Processing Elements (APE) and the Abstract Switch Fabric (ASF). Unique numbers distinguish data Objects. They play local memories role.

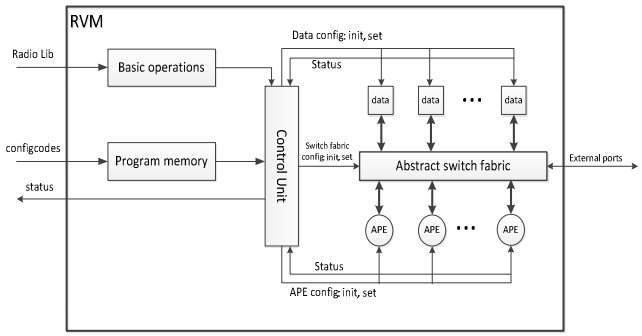

Fig. 2. The Basic RVM.

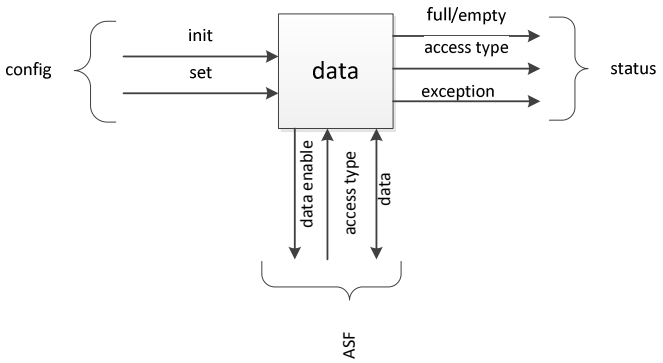

Fig. 3. The Data Object.

The DO structure is represented in Fig. 3. DOs are configured by the instruction **DO_config** with parameters **Init** and **Set.** The **Init** parameter provides immediate values for initializing DOs. The **Set** parameter defines DO attributes such as the DO's size, access cost and the number of read/write ports. DOs communicate with APEs through the ASF by the interface consisting of the data enable signal de, the access type ac ∈ {"read (r)", "read-erase (re)", "write (w)"} and data. The data enable signal (de) is set in logical 1 when a Data Object is full.

APEs convey "access type" (ac) to DO's ports. They dynamically access DOs with read/write operations (ac ="r", "re" or "w") during executing calculations. The DO's content is available for reading ("r" or "re") when de = 1. The empty DO can be written. In this case de = 0. The write operation is blocked when the DO is full i.e. when de = 1. The status interface provides the DO's status information to the CU and consists of state, access type and exception signals. The state signal = 1 if the DO is "full" and 0 in the opposite case, access type = 0 in case of the read operation ("r" or "re") and 1 for the write operation.

The APE structure is shown in Fig. 4. All APEs are numbered. They has the following attributes: the number of ports, the execution cost and the time constraints. The execution cost is any number which might be characterized as execution time, power consumption (average or peak, mW), complexity (number of cycles), MOPS/mW.

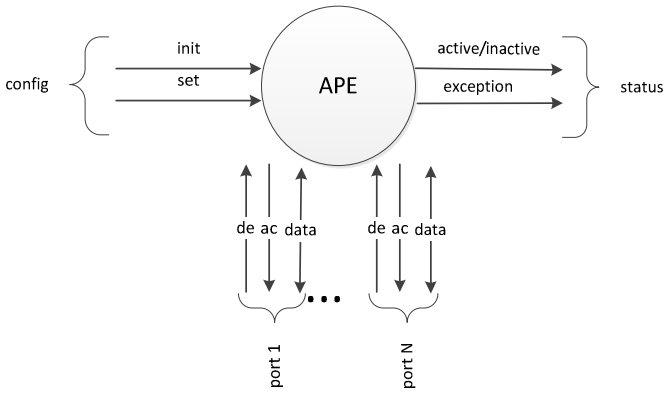

Fig. 4. The Abstract Processing Element.

The time constraint is determined by the worst-case activation time of APE on receiving each task. APE's ports connect the APE to the ASF via data interface lines: de, ac and data. The APE is configured by the instruction **APE_config** with parameters **Init** and **Set.** All APE's ports are numbered. The **Init** parameter brings the op code operation from the Basic Operations. The APE goes to the inactive state with corresponding indication to CU immediately after configuring. The **Set** parameter defines APE's attributes. The APE is uninitialized when is not configured. The APE is active when it has consumed input Data Objects and processes them. The status information is delivered upon operation completion.

The Abstract Switch Fabric connects APEs and Data Objects. One DO can be connected with multiple APEs. One APE can be connected with multiple DOs. The ASF includes data ports (internal or external) and processing ports. The data ports connect the ASF and DOs via interface lines. Data Objects from other RVMs can be connected with ASF through external data ports. Processing ports are connected with APEs via interface lines. The CU configures the ASF by the **Set** and the **Init** instructions. The **Init** instruction creates data and processing ports. The **Set** instruction establishes the internal connectors between data and processing ports and connects these ports with DOs and APEs.

The Abstract Switch Fabric connects APEs and Data Objects. The ASF includes data ports (internal or external) and processing ports. The data ports connect the ASF and DOs via interface lines. Data Objects from other RVMs can be connected with ASF through external data ports. Processing ports are connected with APEs via interface lines. The CU configures the ASF by the **Set** and the **Init** instructions. The **Init** instruction creates data and processing ports. The **Set** instruction establishes the internal connectors between data and processing ports and connects these ports with DOs and APEs.

The Control Unit (CU) generates **Init** and **Set** instructions for APEs, DOs and ASF base on decoding configcodes stored in the program memory. Each configcode includes configuration sections for DOs, APEs and ASF. It also includes additional control flags like Last Configcode (LC), Next Configcode Address (NCA) and optional Next Configcode Address Offset (NCAO). LC = 1 if a configcode is the last code in a task. The field NCAO is augmented if NCA = 1.

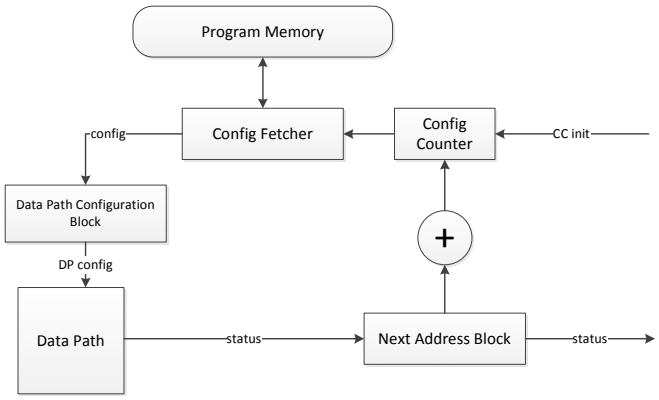

Fig. 5. The Control Unit.

The Control Unit handles task execution and consists of the blocks shown in Fig. 5. The Configuration Counter (CC) points out on the first configcode in a task. The Config Fetcher fetches configcodes from the program memory. The Next Address Block (NAB) calculates the next configcode address. The Data Path Configuration Block configures the data path. CC is setup at the beginning of the task and is updated after the current configcode execution by NAB as CC = CC + |CC| when no exceptions, where |CC| is the length of the configcode with address in CC and CC = CC + NCAO if a configcode flow change is occurred. NAB detects the end of the task after completion of all task configcodes. NAB also generates the status signals, which might be active, inactive, or exception. These signals characterize the RVM status. The RVM is inactive if the Program Memory or the Basic Operations block were not downloaded by configcodes. The RVM is active when it executes calculations.

As it was noted above RVMs might be connected by sharing Data Objects. Therefore, multiple RVMs can create an array of interacting RVMs. Such RVM array executes concurrent asynchronous tasks without control overhead. RVMs can be combined also hierarchically. One or a few RVM might be a part of the bigger RVM. In such case, these component RVMs behaves as the APE and is connected to the bigger RVM ASF by external processing RVM ports. Each component RVM is configured as the APE.

One notes that described RVM is not imperative abstract machine. It is reactive and totally data driven. There are only one Config instruction, which configure data path and after that RVM starts work immediately without additional control. Data Objects are active but APEs are passive. Only data direct and synchronize computations.

IV. Code Portability

RVM abstracts software development from target hardware. Independent software developers are able to create modem software or its separate components without considering particular modem hardware details. By modem software, we assume program code expressing L1/L2 communication algorithms in RVM and Radio Library semantics. Algorithmic operations are taken from the Radio Library and they behave and interacts according to RVM rules. We will call such type of program code as Radio Applications. For example, whole L1 and L2 description of particular Radio Access Technology is considered as a Radio Application.

Radio Application compilation can be broken into two steps as it is pointed out in [6]. During the first step, a front-end compiler translates source codes into RVM configurations (configcode). These configcodes can be executed by the RVM.

During the second step, a back-end compiler compiles RVM configcodes. The result of this step is executable codes, which can be run on a particular target platform. The described approach and the system architecture was approved as baseline architecture for Reconfigurable Radio [2]. Configcodes are stored in a Radio Application Store and can be downloaded into mobile devices by users according to their

needs and radio access availability. Corresponding authority must certify all configcodes before uploading. Mobile devices might have the RVM implemented in either software (as execution environment) or hardware. In this case, the second compilation step can be omitted and configcodes can be interpreted directly on the target HW Radio Platform. Another implementation variant includes the JIT (Just-in-Time) back-end compiler, which translates configcodes during run-time. This variant provides better performance in comparison to the case of interpretation. Both variants require native implementation of the Radio Library for a particular target platform.

On-device compilation in general consumes many resources. It might not be applicable for some mobile devices due to resource constraints. In such case AOT (Ahead-of-Time) compilation is applied but not directly in mobile devices rather outside in some separate server. In this case, configcodes can be preliminary downloaded into a separate server and statically compiled into executable codes. Then they can be downloaded into the mobile device according to user's requests. It will be reasonable if modem vendors provides such services as Vendor's Radio Reconfiguration Service. In this case, all resources for compilation will be located somewhere in vendor's data cloud without consuming mobile device resources.

V. Conclusion

As a conclusion, the ETSI Software Radio Reconfiguration solution represents a highly efficient approach for altering radio characteristics of a Mobile Device through provisioning of software components. The novel Radio Equipment Directive (RED) delivers a corresponding regulation framework, which makes the implementation of the technology possible in the single European market. It is expected that other regulation domains will adopt a comparable approach in the future.

References

[1] ETSI EN 302 969 V1.2.1 (2014-11), Reconfigurable Radio Systems (RRS); Radio Reconfiguration related Requirements for Mobile Devices

[2] ETSI EN 303 095 V1.2.1 (2015-06), Reconfigurable Radio Systems (RRS); Radio Reconfiguration related Architecture for Mobile Devices

[3] ETSI EN 303 146-1 V1.2.1 (2015-11), Reconfigurable Radio Systems (RRS); Mobile Device Information Models and Protocols; Part 1: Multiradio Interface (MURI)

[4] ETSI EN 303 146-2 V1.2.1 (2016-06), Reconfigurable Radio Systems (RRS); Mobile Device (MD) information models and protocols; Part 2: Reconfigurable Radio Frequency Interface (RRFI)

[5] ETSI EN 303 146-3 V1.2.1 (2016-08), Reconfigurable Radio Systems (RRS); Mobile Device (MD) information models and protocols; Part 3: Unified Radio Application Interface (URAI)

[6] ETSI EN 303 146-4, to appear, Reconfigurable Radio Systems (RRS); Mobile Device (MD) information models and protocols; Part 4: Radio Programming Interface (RPI)

[7] ETSI TR 103 087 V1.1.1 (2016-06); Reconfigurable Radio Systems (RRS); Security related use cases and threats in Reconfigurable Radio Systems

[8] ETSI TS 103 436 V1.1.1 (2016-08); Reconfigurable Radio Systems (RRS); Security requirements for reconfigurable radios

[9] DIRECTIVE 2014/53/EU OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 16 April 2014 on the harmonisation of the laws of the Member States relating to the making available on the market of radio equipment and repealing Directive 1999/5/EC, http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:32014L0053

[10] Directive 1999/5/EC of the European Parliament and of the Council of 9 March 1999 on radio equipment and telecommunications terminal equipment and the mutual recognition of their conformity, http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:31999L0005

[11] ETSI TR 102 967: Reconfigurable Radio Systems (RRS); Use cases for dynamic equipment reconfiguration, V1.2.1, 2015

[12] SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION, Joint Tactical Networking Center (JTNC), August 2015, V4.1, available at http://www.public.navy.mil/jtnc/sca/Pages/default.aspx

[13] SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION USER'S GUIDE, Version: 4.1, 23 February 2016 , available at http://www.public.navy.mil/jtnc/sca/Pages/default.as

[14] Wireless Innovation Forum Top 10 Most Wanted Wireless Innovations, Document WINNF-11-P-0014, Ver. V1.0.1, October, 7, 2011

[15] "The Radio Virtual Machine" by M.Gudaitis and J.Mitola III, Oct 31, SDR Forum Workshop, 2000

[16] "The Radio Virtual Machine: A Solution for SDR Portability and Platform Reconfigurability" by Riadh Ben Abdallah, Tanguy Risset and Antoine Fraboulet, IEEE International Symposium on Parallel & Distributed Processing, May 23 – 29, 2009

[17] "Virtual Machine for Software Defined Radio: Evaluating the Software VM Approach" by Tanguy Risset, Antonie Fraboulet, Jerome Martin and Riadh Ben Abdallah, Interbational Conference on Embedded Software and Systems, ICESS, 2010

[18] Acquiring and Sharing Knowledge for Developing SCA Based Waveforms on SDRs, Report RTO MP-IST-092, September, 2010, www.dtic.mil/get-tr-d