A Flexible MAC Development Framework for Cognitive Radio Systems

Junaid Ansari, Xi Zhang, Andreas Achtzehn, Marina Petrova and Petri Mähönen
Institute for Networked Systems, RWTH Aachen University, Kackertstrasse 9, D-52072, Aachen, Germany
Email: {jan, xzh, aac, mpe, pma}@inets.rwth-aachen.de

Abstract—Cognitive radios are becoming the technological foundation for efficiently managing the scarcity of wireless spectrum, fulfilling various QoS demands and allowing different networks to coexist. Cognition and spectrum agility in MAC protocols require adaptability and close PHY-MAC interaction. Classically, MAC protocols have been implemented in hardware, which gives a limited possibility for reconfiguration and customization. Recently, software based MAC implementations have emerged although a close hardware-software co-design is typically required to keep the time critical operations in ASICs or FPGAs. We introduce a MAC development framework for enabling fast composition of MAC protocols, which are best fitted to the application requirements, communication capabilities of the radio, and current spectrum regulations and policies. We decompose MAC protocols into their basic functionalities which are perceived as building blocks and are partitioned across hardware/software. Our framework allows on-the-fly realization of the envisioned MAC protocol through wiring of these fundamental components. By exposing extended metadata and hardware functionalities for the MAC implementation through our granular components, together with the support for run-time re-configuration, spectrum agile and cognitive MAC solutions can be easily realized. In this paper, we describe the design rationale and implementation details of our framework on WARP [1] boards. We show through experimental evaluation that the framework provides flexible means for prototyping different reconfigurable cognitive and spectrum agile MAC protocols.

I. INTRODUCTION

There has been a surge of wireless technologies over the last decade, which has led to the crowding of existing spectrum. In order to address the resulting congestion and scarcity in the shared wireless spectrum, cognitive MAC protocols are designed following spectrum management ideologies. Intelligent management of spectral resources, cooperation among nodes and advanced sensing in medium access procedures require adaptability and close interaction between PHY and MAC layers. Adaptability and reconfiguration based on network statistics and channel conditions are needed during the execution of a cognitive MAC in order to meet the application requirements.

MAC protocols implemented in hardware (e.g. IEEE 802.11 NICs, Bluetooth dongles, etc.) leave limited room for re-configuration and customization. These static and rigid hardware based MAC implementations are suboptimal for fulfilling the diverse and changing QoS application demands and thus, fail in flexible spectrum management domains [2]. In recent years, software based MAC implementations have emerged but these have shortcomings in meeting time critical deadlines. Software Defined Radio (SDR) platforms have demonstrated possible advantages in the MAC designs based on the flexible radios. However, despite the fine-grained access control knobs offered by the platform, current MAC development frameworks on SDR platforms lack the support for flexible MAC design and fast on-the-fly re-configuration.

In this work, we define a set of fundamental MAC functionalities as a library so that a wide range of MAC protocols can easily be realized by simply combining these functional blocks in an appropriate manner. Cognitive MAC protocols can benefit from a rich and fine-grained set of exposed radio parameters in our framework. The building blocks are implemented in hardware or software depending on the associated communication and computational overhead and the degree of their reuse across different MAC implementations. A Wiring Engine is designed in order to bind individual MAC blocks together and to coordinate the data and control flow among the blocks. It allows composition of MAC protocols at runtime and therefore enables fast MAC reconfiguration. We have demonstrated in [3] that our framework implementation on WARP [1] boards allows rapid prototyping of spectrum agile cognitive MAC protocols with high degree of code reuse and performance characteristics.

The rest of the paper is structured as follows. Section II provides the related work by other research groups towards flexible MAC development and testbeds. In Section III, we identify the main design goals and requirements for re-configurable MAC designing in cognitive radios. Section IV discusses the design rationale of our framework while Section V describes the implementation details on WARP boards. In Section VI, we present the experimental evaluation results and finally conclude the paper and give future directions of our work in Section VII.

II. RELATED WORK

MAC protocols are typically implemented in a monolithic fashion with tight coupling to the underlying hardware. A hardware specific implementation restricts the reconfigurability and flexibility aspects needed by spectrum agile and cognitive MACs. Bianchi et al. discuss the advantages of an adaptive and programmable MAC framework, which satisfies time varying QoS demands [4]. In order to address the issue of flexibility and adaptability, three main approaches have been investigated. MultiMAC [5] allows switching among a few pre-defined standalone MAC solutions to suit the changing
network and application requirements. This approach has limited scope because it can only find an approximate “best-fit” solution from a limited number of protocols in the MAC pool. In order to enhance MAC performance, a few modular designs [6] have been proposed. These designs are inspired by the component oriented and non-classical view of protocols by Braden et al. [7]. Unfortunately, many of these designs either lack the actual implementation or are incapable of satisfying the strict real time requirements of MAC protocols. In order to provide reconfigurability features to a MAC developer through flexible interfaces, Sharma et al. propose FreeMAC [8], which is a multichannel MAC development framework on top of the standard IEEE 802.11 hardware. FreeMAC aims at supporting frequent channel switching and efficiently controlling the timings for packet transmissions. A similar work has been demonstrated in [9]. In [10], the authors control the timings of the frames through the MadWiFi driver for Atheros based NICs in order to improve the overall throughput. In [11], authors show a TDMA MAC protocol implementation in software (Linux user-space) using the MadWiFi driver for Atheros AR5212 chipset. However, since IEEE 802.11 hardware is restricted in providing accessibility to the radio and PHY parameters as are needed by spectrum agile and cognitive MACs, the configurability aspects of these protocols are limited. Furthermore, these designs are not implemented in a modular fashion; only certain MAC parameters can be changed instead of reconfiguring MAC composition and its functionality.

Pure software MAC implementations, although extremely flexible, fail to meet the time-critical deadlines. A fairly comprehensive discussion on the shortcomings of purely software MAC implementations can be found in [2]. Some of the low-cost SDR-platforms do not offer good support for flexible MAC implementation as has been learnt by other research groups in the case of USRP and GNU Radio platforms [2], [12]. In contrast, WARP provides wider experimental opportunities for customizing the hardware and prototyping MACs. In [13] authors have implemented a fully distributed cooperative MAC/PHY system on WARP boards with accurate timing characteristics.

III. DESIGN GOALS

Hardware-software co-design: Purely software implementations often fail to meet the time-critical deadlines, while hardware implementations tend to be rigid, lack the room for experimentation and are unable to provide the flexibility and re-configurability needed by cognitive MACs. Therefore, HW/SW co-design approaches and hybrid processor FPGA designs such as Garp [14] becomes desirable. Portability and code reuse: Recently, efforts have been made for implementing efficient and flexible radioPHY abstractions. Existing MAC implementations are carried out mostly in monolithic fashion. Therefore, it is important for a development framework to provide MAC abstractions with well-defined interfaces, which allow code reuse across different MAC implementations.

Granular parameters: MAC protocols for spectrum agile cognitive networks require extended and fine-grained access control to different PHY and MAC parameters [15]. These hooks are needed to be exposed through a rich set of APIs in a MAC development framework.

Run-time reconfiguration: One of the key characteristics of cognitive MAC protocols is the ability to adapt its behaviour to the changing spectral needs and network conditions. MAC designs and development frameworks should therefore support mechanisms for fast on-the-fly reconfiguration.

IV. FRAMEWORK DESIGN

MAC protocols show many functional commonalities. In this work we analyze different protocols based on CSMA/CA, TDMA and hybrid principle in order to identify the common functionalities so that different protocols can be implemented using the same set of building blocks. In order to bind the blocks together for realizing a particular MAC, we design a tool called as the Wiring Engine (Please refer to Section IV-B for details). This engine coordinates the flow of data and control among different blocks and schedules their execution sequence. It also allows composing and modifying MAC behaviors at runtime.

A. Decomposition of MAC Protocols

We define a list of basic functional blocks as a library so that a wide range of MAC protocols can be designed using these building blocks. MAC functionalities such as timer, backoff counter, carrier sensing, frame formation, sending a frame, receiving a frame, etc., are not only common to different protocols but are often also repeated within a particular protocol realization. Additionally, access to radio control parameters like switching the state of the radio (transmit, receive, sleep), setting the transmit power level, tuning the receiver sensitivity value, selecting the operating frequency, etc., are also typically needed by different MAC implementations. We have shown that a library of these basic components available to a user through a GUI together with an auto-code generation tool allows rapid prototyping of MACs [16]. As an example, Table I lists the commands exposed in the Timer block.

The pattern in which various primary blocks are connected is identical across different MAC implementations. Therefore, we define secondary level MAC building blocks, which are

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>start()</td>
<td>starts a timer</td>
</tr>
<tr>
<td>stop()</td>
<td>stops a timer if the timer is not expired yet</td>
</tr>
<tr>
<td>suspend()</td>
<td>suspends the running of a timer until it is resumed</td>
</tr>
<tr>
<td>resume()</td>
<td>resumes the running of a timer after the suspension</td>
</tr>
<tr>
<td>getStart()</td>
<td>returns start time of the timer</td>
</tr>
<tr>
<td>getDuration()</td>
<td>returns timer duration</td>
</tr>
<tr>
<td>getNow()</td>
<td>returns current time</td>
</tr>
<tr>
<td>getStatus()</td>
<td>returns the status (RUNNING, SUSPENDED, STOPPED)</td>
</tr>
</tbody>
</table>

TABLE I

TIMER INTERFACE
TABLE II

<table>
<thead>
<tr>
<th>Block</th>
<th>Usage and the composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Backoff</td>
<td>Random backoff mechanism</td>
</tr>
<tr>
<td>Expecting Frame</td>
<td>Used when the node is waiting in anticipation of a packet</td>
</tr>
<tr>
<td>Send Packet</td>
<td>Called after seizing a channel free</td>
</tr>
<tr>
<td>RTS/CTS/DATA/ACK</td>
<td>Four-way handshake mechanism</td>
</tr>
</tbody>
</table>

composed of basic blocks. In the following, we describe in detail with an example of Random backoff, how a secondary level block is realized in our framework. The random backoff functionality is very common in CSMA/CA based protocols and consists of the primitive blocks Timer, Carrier Sensing and Random Number Generator (RNG). Fig. 1 illustrates the realization of Random backoff block with the associated inputs, outputs and execution flow. The Random Number Generator block takes Range and Precision as inputs for the random number generation, which is used to start a ONE_SHOT timer. If Carrier Sensing (CS) is disabled, a Null waiting state is assumed till the timer expires and the block exits. Otherwise, CS is performed for a specified duration. If CS returns true, the timer is suspended and CS is performed again. However, if the output is false, the timer is resumed and CS is repeated. On the other hand, if the timer expires during the operation, the block exits. The figure shows a detailed control flow (sequence of execution and preemptions) in the Random Number Generator block. We have realized different secondary MAC blocks in our framework based on the same principle. Table II lists the most common of these blocks. The secondary level blocks can also be used as sub-blocks for other complex blocks.

Fig. 2 illustrates the realization of IEEE 802.11 DCF using the fundamental and secondary blocks. The details on data and control flow are omitted to keep the figure simple. The unfilled blocks represent the binding logic and decision expressions, which determine the state-machine of the MAC. The state-machine logic is controlled by the Wiring Engine. It may be noted that certain blocks, for instance Send Packet, Timer and Radio Switching, are repeated within the MAC realization.

B. Wiring Engine

The approach of binding components through the Wiring Engine widens the possibilities for different MAC realizations. Our approach also enables dynamic adaptation of MAC behaviors to the changing environmental conditions and application requirements. The concept of wiring component based systems has been long established where a software waveform can be expressed as a network of boxes communicating with each other via connecting buses. Compositional adaptation techniques have been well investigated [17] because of the need for run-time reconfigurations, such as in robotic software systems. We have developed a lightweight Wiring Engine based on the usage of function pointers. Any potential race condition is avoided using pre-defined dependency tables for each MAC component. The construction and execution path of a state-machine is dynamically redirected through modification of function pointer associations. This approach enables both runtime reconfiguration of MAC protocols according to pre-defined rules within the framework, and on-the-fly realization of user configured protocols. The fundamental blocks are wrapped with a standard API: int Function(int, radio), where the input parameters to the block includes an integer, which is the output from the previous block, and a pointer to a radio object which contains all the radio and control parameters for the block. For example, carrier sensing block and send packet block use APIs, int CarrierSense(int para, radio r) and int SendPacket(int para, radio r), respectively. A function pointer is defined as typedef int (*FuncPtr)(int, radio). A doubly-linked list is used to maintain the elements used in the protocol since both ‘insert before’ and ‘append after’ functionalities are re-
required for fast configuration. Fig. 3 gives a very simple example of how the linked list can be used. The function `list_initialize()` initializes the list and includes the default elements. For example, MAC blocks like `SendPacket` can be defined to be used as default. The function `list_execute()` executes each and every element in the list in a sequential manner. The Wiring Engine allows run-time user specified modifications. A user can specify add/remove elements from the currently running protocol interactively (over the serial or Ethernet port) and also start/stop/jump in the configuration. For example, when a user inputs `add_carriersense`, an entry for the linked list is created with element `CarrierSense` and inserted to the list according to pre-defined block dependencies, i.e., `CarrierSense` should be inserted before `SendPacket` block. The inter-dependency among blocks is kept in a table. There are three possible relationships between any two blocks: 'insert before', 'append after' and 'independent'. Our approach of injecting code blocks in the already executing code is superior over Multi-MAC [5] scheme as it is not limited to the number of MACs in the pool. Besides user triggered reconfiguration, our Wiring Engine also facilitates run-time auto-adaptation of the MAC protocol based on the same principle as described above. A set of pre-defined rules can be used to modify the MAC protocol. For example, when the neighbourhood size increases, `BackOff` before `SendPacket` is desirable for reducing the collision probability. In this case, `BackOff` is automatically inserted into the list. Simplifying construction of a MAC protocol through binding a limited set of blocks advocates the design of a user-friendly meta-language MAC compiler. We have defined a MAC specific language, where a user can construct a protocol through minimum lines of code. A compiler translates the domain specific language into a GCC compliant code, which is then compiled and downloaded onto the target platform. The compiler also checks the implementation logic and report errors like the violation of inter-block dependencies, etc. In the future, we will target many-core processor architectures for the implementation of the MAC blocks and the language semantics will also allow the Wiring Engine to parallelize and prioritize the processes based on their dependency tables.

V. PROTOTYPE IMPLEMENTATION ON WARP BOARD

We have implemented our framework on WARP boards [1] using the OFDM Reference Design v.14. All the fundamental MAC blocks in the framework are implemented in the FPGA, except the random number generator (RNG), which is implemented in the software on PowerPC. The virtualization of the fundamental blocks through flexible wrapper APIs and the Wiring Engine implementation is carried out in software running on PowerPC. In our framework, standard components such as timers, memory resources, serial interfaces and GPIOs are combined with the state-of-art radio control. We also expose an extended radio functionality to users such as setting the transmit power levels, channel selection, receiver sensitivity thresholds, setting modulation schemes, etc. These additional hooks facilitates cross-layer designing. For instance, we lavish from a close PHY-MAC interaction in the implementation of CogMAC [18] using the framework. In order to avoid polling delays, we have modified the basic hardware reference design to include an interrupt controller for all the components available to the framework. Events such as the reception of a packet are combined with user-defined ISR functions. The interface for each object is standardized to ease the debugging process. Particular hardware requirements include, for instance, granularity of timers or type of modulation schemes, etc., which may be defined on initialization or changed at the runtime, when required.

Code portability and ease of MAC designing are achieved through our high-level hardware abstraction architecture as shown in Fig. 4. More complex MAC designs can be envisioned on the platform, where the number of required resources of a particular type exceeds the number of hardware components. The framework user is in this case exempted from the resource management tasks as the framework keeps state information upon freeing the resources. Timers are a typical example of such scheduled objects. In the User MAC Code, timer objects may be generated by specifying object properties - a timer may either be periodic or fires only once, the granularity of the timer is in the scale of milli- or microseconds. The Framework Object Interface manages those objects and destroys them upon expiration of their lifetime. Instead of assigning a particular hardware timer on the WARP, the Virtual Component Code only updates the Object State Information. For this reason, an update to the duration of the timer object causes an update in the object state information. Only if inherently hardware-dependent operations such as starting the timer are requested by the user, the hardware scheduler allocates real resources. Interaction with those resources is carried out through platform-specific drivers that interact with the available hardware. Other resources in our implementation are RNG and CRC units. In classical designs for adaptive MAC frameworks, the entire hardware platform is reset upon changes of the code. On the other hand, block-wise resetting allows for a quicker adaption of the new code and lowers
black-out times of the hardware in our framework. This object-
persistency is supported in the framework by introduction of
a split-phase resource initialization.

VI. EXPERIMENTAL EVALUATION

In this section we describe the code reuse across different
MAC implementations on our framework and the reconfigu-
ration response time and memory footprint of our prototype
implementation. We have also considered different GCC compi-
ler optimizations for code efficiency in our measurements.

A. Component Reuse

Table III lists the MAC block reuse in realizing various
protocols on the WARP boards. The gray shaded blocks are
the primary blocks. It can be observed that certain blocks
are used multiple times within a particular protocol, which
advocates the idea of realizing the key atomic functionalities
in the hardware for higher computing and communication
efficiency. From the perspective of a user, a MAC becomes
simpler to implement if the framework provides support for
secondary level blocks. We have realized CogMAC [18], a
decentralized spectrum agile cognitive MAC protocol with
fairly complex multichannel operation in a fast and easy way
using the framework.

B. Code Size and Memory Footprint

The Wiring Engine is implemented in the software on the
PowerPC core on the WARP board, therefore we analyze the
ELF (Executable and Linkable Format) binary targeted for
PowerPC. Since compiler optimizations directly influence the
memory use [19], we analyze ELF files generated with differ-
ent optimization options. Table IV lists the memory footprint
break-down for a pure CSMA MAC protocol developed using
our framework without the Wiring Engine in a monolithic
manner (left) and using the Wiring Engine (right). Please note
that the memory usage of the Wiring Engine depends upon
the number and type of blocks used in a particular MAC
and its state-machine logic. Therefore, we consider a pure
CSMA MAC realization (with only three blocks) with and
without using the Wiring Engine for our analysis. It can be
observed from the table that the Wiring Engine imparts only
a slight memory overhead in the uninitialized data contributed
to the program memory (.bss) and the code size (.text). Low memory foot-print makes the framework also relevant
to the domains of component-based network implementations
and cross-layer designing in relatively resource constrained
embedded systems.

C. Response Time Overhead

A fast response time is achieved through the function
pointer based implementation approach followed in the Wiring
Engine. Fig. 5 shows the average time required for a MAC
to reconfigure itself. The graph shows the average results
for 10,000 readings at different optimization levels. Here we
consider the elapsed time for the Wiring Engine to update the
state-machine of the MAC upon receiving a new configuration.
The bar graph also shows the percentages of the individual
reconfiguration delays. Please note that since the execution
time for different blocks is different, here we consider the case
when a configuration is updated (block(s) removed or inserted)
instantly without waiting for the completion of the currently
executing block. In other words, the currently executing block
has an ‘independent’ entry in its dependency table. Our results
show a fast reconfiguration response, which makes it suitable
for designs with time-critical requirements.

In addition to the auto-reconfiguration response time, we
also measured the latencies associated with user-driven con-
figuration updates of the framework implementation on WARP
boards. Fig. 6 shows the average response time for different
number of user triggered re-configuration operations (insert,
remove, change the order). The user updates are sent from
the PC to the WARP board over the UART interface. Please
note that for all the optimization levels, the required re-
configuration time shows a linearly increasing behaviour with
increasing number of re-configuration operations.

<table>
<thead>
<tr>
<th>Block</th>
<th>AlohA</th>
<th>PureCSMA</th>
<th>B-MAC</th>
<th>IEEE 802.11</th>
<th>S-MAC</th>
<th>CogMAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendPacket</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>ExpectFrame</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>RTS/CTS/DATA/ACK</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Random No. Gen.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Timer</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Carrier Sensing</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Channel Switching</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Radio States</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SendFrame</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ReceiveFrame</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table III: Block reuse for MAC realizations on WARP.

Table IV: Memory footprint (in octets) without and with Wiring Engine.

<table>
<thead>
<tr>
<th>Level</th>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt 0</td>
<td>109750</td>
<td>1704</td>
<td>17812</td>
<td>129266</td>
</tr>
<tr>
<td>Opt 1</td>
<td>97246</td>
<td>1704</td>
<td>17796</td>
<td>116746</td>
</tr>
<tr>
<td>Opt 2</td>
<td>98226</td>
<td>1704</td>
<td>17796</td>
<td>117726</td>
</tr>
<tr>
<td>Opt 3</td>
<td>102914</td>
<td>1704</td>
<td>17796</td>
<td>122414</td>
</tr>
</tbody>
</table>

![Fig. 5. Response time for MAC auto-configuration on WARP boards.](image-url)
D. Software versus Hardware Implementation

We argue that implementing a MAC protocol purely in software fails to achieve timing requirements and therefore results in poor performance characteristics. On the contrary, a hardware implemented MAC, e.g., IEEE 802.11 on a COTS NIC, though optimized for the design, does not permit flexible experimental room and the needed PHY-MAC interaction. We approach this problem by implementing selective basic functionalities in hardware while providing flexible APIs and virtualization in the software (c.f. Section V). This enables to simultaneously achieve hardware acceleration without compromising on flexibility. A one-to-one comparison in terms of the response time and execution speed (which directly affects the latency and throughput as shown in [8]) of a MAC implementation on a WARP board with an implementation on a standard IEEE 802.11 NIC or with a GNU radio [20] based software implementation is unfair. Therefore, in order to study the benefits of implementing the key atomic functionalities in hardware, as examples, we carried out the implementation of CRC and RNG blocks both in a custom hardware as well as in the software on the ARM926 core. We have observed a performance gain of ca. 2041% and 779% in the speed of the CRC computation and a random number generation through the hardware implementation, respectively.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the design details of a flexible MAC development framework based on the MAC decomposition philosophy. The key features of our framework are its modular design and support for run-time re-configuration. The Wiring Engine allows rapid on-the-fly reconﬁgurability, which is an important requirement for spectrum agile and cognitive MAC solutions. We expose granular blocks through flexible functional abstractions in order to provide a closer access to the PHY/MAC controlling parameters than contemporary closed MAC development frameworks. This certainly widens the experimental room for implementation of cross-layer designs. We have prototyped our framework on WARP boards with custom modiﬁcations of the hardware on FPGA for performance enhancements. The evaluation results on WARP boards indicate that our framework enables rapid MAC prototyping. Measurement results indicate that using the function pointer and dynamic dependency table lookups keep the latency of reconﬁguration low. Our work in progress includes the MAC language and meta-compiler tool chain for multilcore platforms. We are also planning to have a possible public release of the code for interested parties.

Acknowledgment

We would like to thank partial ﬁnancial support from RWTH Aachen University, DFG through UMIC research centre (Nucleus project) and EU through 2PARMA project.

References