An Infrastructure for Flexible Runtime Reconfigurable Multi-Microcontroller Systems

Claudius Stern, Philipp Adelt, Matthias Schmitz, Lisa Kleinjohann, and Bernd Kleinjohann

C-LAB, University of Paderborn, Paderborn, Germany
claudis@c-lab.de,
Home page: http://www.c-lab.de

Abstract. Embedded systems in robotics or mechatronics need flexibility since they are working in dynamic environments. We consider an embedded modular multi-microcontroller system. Each module includes a microcontroller and special purpose hardware like a motor driver. Usually a change of the embedded software necessitates a direct access to all the devices (microcontrollers) to reload the code.
To overcome this disadvantage we introduce an infrastructure for flexible runtime reconfiguration of microcontroller modules within a system. The infrastructure enables the system to be coarse-grain reconfigurable on module level from one single point of access.
By using our infrastructure the system can remain operational during reconfiguration except the modules that actually get reconfigured. The infrastructure can cope with hardware changes during runtime like disconnection and reconnection of system parts.

1 Introduction

Embedded systems in robotics or mechatronics need flexibility since they are working in dynamic environments. We consider an embedded modular multi-microcontroller system whereby each module includes a microcontroller and special purpose hardware, e.g., a motor driver. Typically these systems are rather complex in terms of number of microcontrollers or in terms of communication structure. Nowadays those system are statically built so that an adaptation to a new environmental situation often requires a complete rebuilding of the entire system. Of course, for improvements of the algorithms used in embedded microcontrollers a reconfiguration of the according controller is also needed. If possible at all, a change of the embedded software on the embedded devices often is very difficult because a direct access to each device is necessary. In such cases a solution would be needed that provides one single point of access for reliable reconfiguration of the embedded software of a microcontroller module which is deeply embedded in the system.

As an example consider a complex mechatronic system with more than hundred microcontrollers. Given that about 20% of the microcontrollers do the same job, like motor controlling, at least 20 microcontrollers have to be reconfigured if

an error is revealed or an update is necessary. Hence, besides the single point of access it would be useful to have the ability not only to reconfigure a single microcontroller but to reconfigure a functionally identical group of microcontrollers simultaneously.

Now consider a reconfigurable mechatronic system, e.g., a truck which could be equipped with different types of accessories. Each accessory for its own is a mechatronic system which is equipped with multiple microcontrollers. If, e.g., a company with several of these multi-use trucks finds an error in the height-level control of the snow plow accessory, a reconfiguration is necessary. An easy way to reconfigure these microcontrollers would be to plug a reconfiguration device in each truck equipped with the faulty snow plow. If for some reason this reconfiguration device is plugged in a truck without a snow plow the system should be able to identify its actual structure and the currently available functional units to prevent wrong reconfiguration. Furthermore, considering a necessary reconfiguration which would reconfigure microcontrollers both on an accessory as well as on the truck itself, the system should be able to reconfigure the one part without having the other part available.

As third example let us regard a production line as a mechatronic system. It could be very expensive to stop the complete system. To consider a reconfigurable system to be installed, a reconfiguration must not lead to a complete system stop. One possibility is to split the production line into sections—which already is common practice—where each section can be stopped individually. The naïve approach would bring up the non-central reconfiguration issue again. Splitting up the production line while keeping the central reconfiguration would need an architecture which ensures the reconfiguration process not to interfere with the functionality of the rest of the system.

The examples described above show that a flexible modular approach is necessary for system reconfiguration. The individual modules should represent functional units which can be combined with mechanics to create mechatronic functional units, e.g., a driving unit for robotics. The system has to be flexible enough to be used at several different places in a robotic system, e.g., as a motor controller or as a multi-servo controller. Components have to be reconfigurable without the necessity of accessing them directly. They should be reconfigurable during runtime while the rest of the system remains operational. The user should not be bothered with details of reconfiguration. The system should be able to identify its own structure.

We envision to plug a system together and when connected to a PC, a diagram of the functional unit structure appears. The user then would be able to select those parts of the system he wants to reconfigure.

In this paper we present an infrastructure for a flexible runtime reconfigurable microcontroller system, that shows the following features.

- Single point of access for reconfiguration:
  Our infrastructure provides reconfiguration-access to the complete system via a single point of access.

- Multicast reconfiguration:
  Simultaneous reconfiguration of multiple modules is possible.
- System enumerates nodes automatically and unambiguously:
  Independent from user-space communication, the system can determine its own structure and the system assigns unique identifiers automatically to its nodes.
- System recognizes changes:
  When a node is removed or added, the system recognizes this situation and starts a new enumeration cycle automatically.
- Separation of concerns: communication vs. reconfiguration:
  The user needs not to bother about matters of reconfiguration.
- Reconfiguration can be independent from user communication channel:
  We integrate two independent communication domains for reconfiguration and user-space communication. It can be selected which communication channel is to be used for reconfiguration.
- Runtime reconfiguration on module level:
  The reconfiguration of a module is possible without stopping the whole system. Only the module itself has to be stopped.

The remainder of this paper is organized as follows. In Section 2 we describe the underlying architecture of our reconfiguration infrastructure. In Section 3 we focus on the automatic structure determination. In Section 4 the reconfiguration process is described. Section 5 then shows the application of our infrastructure to a demonstration system. After that, we discuss related work in Section 6. Finally, we conclude our work in Section 7.

2 Reconfiguration Architecture

Our intention was to create a modular system of microcontroller boards for control purposes. A common problem nowadays is that each individual microcontroller is reconfigurable, but only at its own connector. Now imagine a large system with 10 or more microcontrollers deeply embedded within the system. Here a single point of access to connect to the system is desirable even if only one microcontroller has to be reconfigured. Therefore we designed modular microcontroller boards and enhanced this design with a reconfiguration architecture. The resulting infrastructure is depicted in Fig. 1.

In the following we use the term *node* for modules which have a specialized logic and a dedicated microcontroller for reconfiguration and structure recognition. The architecture distinguishes three basic types of modules.

- Communication nodes
- Execution nodes (e.g., I/O nodes, calculator nodes)
- Power supply modules (no node logic)

Communication nodes hold a central position within the infrastructure. They are the bridge between the embedded system and a controlling infrastructure. For

this purpose one Communication node is equipped with a microcontroller C_{comm} and several internal and external communication interfaces. Hence a Communication node acts as the single point of access to the system.

Execution nodes are specialized microcontroller-driven devices which are used for the actual control tasks like motor controlling, waveform generation, etc.. Each Execution node has a main microcontroller C_{main} for the actual control task and a dedicated reconfiguration microcontroller C_{config} for reconfiguring C_{main}.

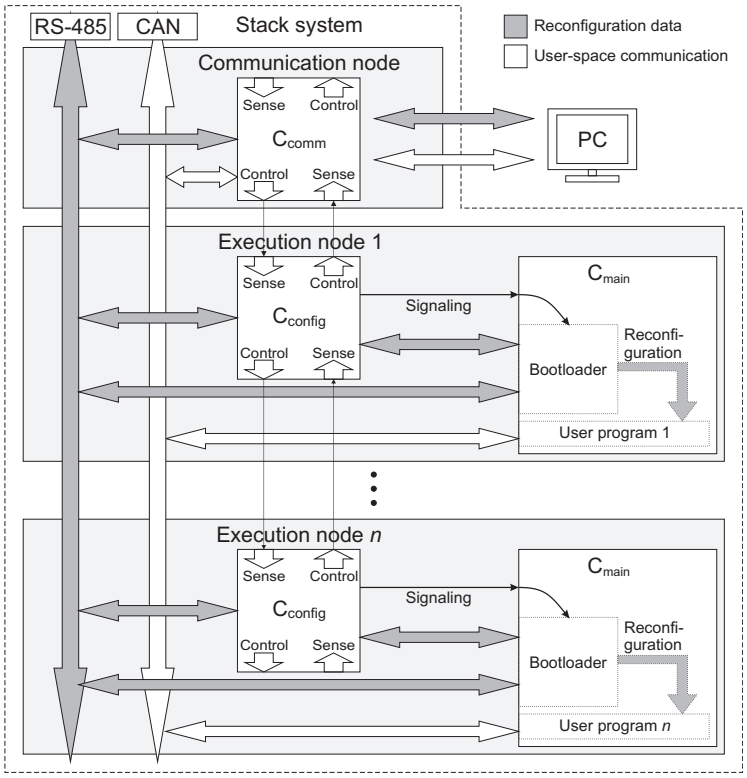


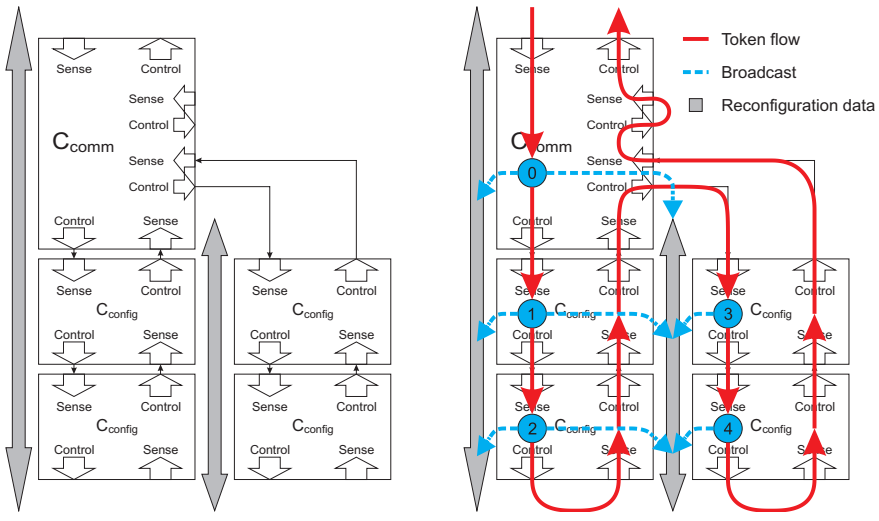
Fig. 1. Reconfiguration Architecture

Fig. 1 shows two independent communication channels throughout the system (CAN in white, RS-485 in gray). In this case the CAN bus is used for user-space communication (although it could be also used for reconfiguration) and the RS-485 bus is exclusively used for reconfiguration. The ports *Sense* and *Control* which are shown in the figure are used for the automatic structure detection. They are important during the system's initialization and whenever the system is structurally changed, e.g., because of parts of the system being switched off or on. The automatic structure determination of our architecture

which is described detailed in Section 3 enables the reconfiguration of systems whose hardware structure can be changed. The separation of concerns between user-space communication and reconfiguration enables a system reconfiguration during run-time. While one microcontroller of the system is reconfigured, the rest of the system may remain operating.

3 Automatic Structure Determination

In order to increase the flexibility of a reconfigurable microcontroller architecture, such a system should be able to determine its own physical structure and to recognize changes to this structure. Fig. 2 depicts a simplified block diagram of the structure of a stacked system. Actually, two connected stacks are shown. Only one of them is equipped with a Communication node. However, in some cases it could be useful to have more than one Communication node in the system. Since Communication nodes act as a bridge between the internal communication architecture and an external one, more than one Communication node could be needed. If the device which uses the system is, e.g., equipped with an internal control PC, one Communication node would be connected to this internal PC and another one would act as the reconfiguration access point. Another example for a scenario with more than one Communication node would be a setting where multiple points of access to the system are desirable, e.g., at the front and the end of a large production line.



(a) Daisy chain; stack with external nodes (b) Enumeration sequence

Fig. 2. Exemplary system configuration with enumeration

Unlike in Section 2 here only that part of the system is of interest which is concerned with the actual reconfiguration. Hence, only the Communication node, C_{comm}, and the reconfiguration microcontrollers of the Execution nodes, C_{config}, are shown instead of the entire nodes. While determining the structure of the system, every node gets a unique number which is later used as an address for the reconfiguration process (Section 4). The structure determination uses a depth-first search which is also used to assign the addresses at the same time.

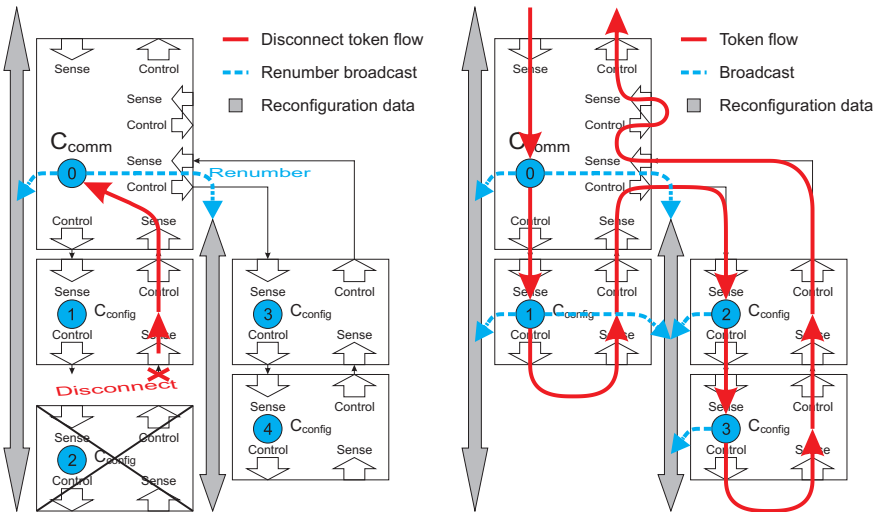
We now describe an exemplary enumeration on the basis of the configuration shown in Fig. 2(a). Fig. 2(b) depicts the enumeration sequence and parts of the communication. Each node, except the Communication node, has two signaling ports to its logical top and the same to its logical bottom. Note another difference to Fig. 1. C_{comm} is equipped with two additional signaling ports where external stacks can be connected (see Fig. 4). *Sense* is an input port used to recognize a connected node and *Control* is an output port used to send signals to a connected node. Both signals have a predefined signal level (Control = 1, Sense = 0). Hence, initially each node is able to detect whether another node is connected. When two nodes are connected, the Control port pulls the Sense port of the other node to 1.

We defined the root node to be the one that initially has no node at its top. The initial 0 at the top Sense port is regarded as a token signaling to the root node that no other node is on top of it. In Fig. 2(a) the root node is C_{comm}. Note that any of the nodes could take this position.

All nodes hold a variable where the currently highest node address is stored (initially -1). C_{comm} gets the token (its top Sense port is 0) and therefore may take an address. It takes address 0 and broadcasts this through the system. After a node has taken an address, it then passes the token to its children. When the token returns through the bottom Sense port, it has to be passed to the next child or—if all children returned the token—it has to be passed to the parent node.

After C_{comm} has broadcasted its address, C_{comm} disables its bottom Control port (passes the token to its first child). This causes a change on the Sense port of C_{config} below C_{comm}. This signals the node below node 0 that it could now take the next address. This process is continued until no node is connected at the bottom port. In the exemplary case of Fig. 2(b) the last node in the first chain is node no. 2. It detects that no further node is connected and therefore passes the token to its parent (node 1). This signals node 1 that all nodes below have finished their enumeration phase.

Node 1 then passes the token to node 0 (C_{comm}). Node 0 recognizes the return of the token and passes it to the next child. In this case the child is the first node of the external stack. It takes the address 3 and passes the token to the next node. After node 4 has taken its address, the token is returned to node 3 and then to node 0. C_{comm} recognizes that it has no further children to pass the token to. As C_{comm} is the root node it then can broadcast the end of the enumeration phase. All nodes then return their Control ports into the initial state which enables the system to recognize changes.



(a) Disconnect event (b) renumeration sequence

Fig. 3. renumeration after disconnect event

After a change of the system's structure a renumeration has to be initiated. Fig. 3 depicts the two stages of the renumeration process exemplaryly for a disconnect event. Since all nodes have reset their Control ports, node 1 in Fig. 3(a) is able to detect the disconnection of node 2, as the Sense port of node 1 switches from 1 to 0. This level change is regarded as a token signaling a disconnect event to the sensing node. The token is passed to the parent until the root node has been reached. Node 0 then sends a broadcast signaling the upcoming renumeration. After that a normal enumeration process starts as shown in Fig. 3(b).

4 Reconfiguration Process

As we mentioned before, we envision a pluggable modular system whose structure is displayed when connected to a PC. After the system has determined its structure, every node has a unique address and is therefore able to reconfigure itself. Besides the two basic types Communication node and Execution node, the system is also able to distinguish between different nodes of the same basic type. The user of such a system can define groups of functionally identical nodes, e.g., motor drivers. In a structural diagram the user then would be able to identify and to select a single node as well as a group of nodes for reconfiguration. The usual reconfiguration should follow the scheme described below:

– C_{comm} of the Communication node, which wants to start a reconfiguration process, sends a message over the reconfiguration bus. The message is addressed to the C_{config} microcontrollers of those nodes that have to be reconfigured and contains the following data:

  - Desired function, e.g., *enter programming mode*
  - Desired channel for further communication, e.g., *RS485*
  - List of addressed nodes
- The reception of the initial message will be confirmed by all selected C_{config} microcontrollers. The confirmations will be sent in the order of the address list.
- All selected C_{config} microcontrollers then put their C_{main} into reconfiguration mode and prepare them for the communication with the desired communication channel.
- After C_{main} has entered the reconfiguration mode, it transmits a confirmation to the initiating C_{comm} using the selected communication channel.
- C_{comm} has to wait for all confirmation messages, which are sent over the selected communication channel.
- For every memory page to be transmitted, three types of messages are exchanged.
  - A *start message*, containing information about the content type, e.g. *EEPROM*, *FLASH*, the start address or page number, the number of expected data messages and a checksum for the complete data to be received.
  - *Data messages*, containing the actual data. Dependent on the used communication channel, these data messages can have different sizes and may be also protected by a checksum.
  - A *finishing message*, causing all recipient C_{main} microcontrollers to check the data for completeness and correctness and to write the received data to the according memory. This ensures that the reconfiguration process starts only, if all data have been received correctly. If one of the recipients reports an error, the process is restarted. Data which already have been received correctly will be ignored, so that only the erroneous nodes are reconfigured again.
- As last step, C_{comm} sends a finishing message to all selected C_{config} microcontrollers, causing them to reset their C_{main} microcontrollers into normal operation mode.

Using this communication scheme for reconfiguration ensures that single nodes as well as a group of nodes can be reliably reconfigured.

5 Realization of the Demonstrator

Fig. 4 depicts an exemplary setup of our stack system. The system consists of a Power supply module, one Communication node and two Execution nodes. The Communication node can be easily identified as it is equipped with a USB connector and two external ports to connect to other stack systems. The Power supply module has an according external connector to get connected with another stack system.

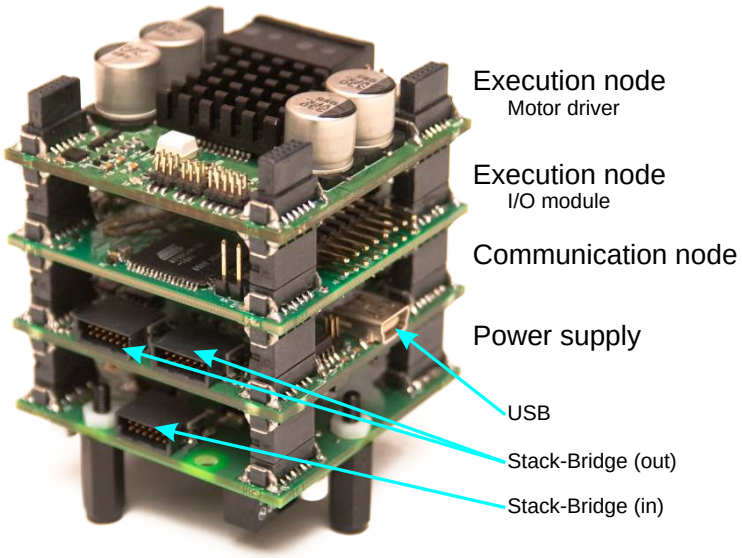


Fig. 4. Stacked sytem

The Communication nodes hold a central position within the infrastructure. They are the bridge between the embedded system and a controlling infrastructure. For this purpose one Communication node is equipped with several internal and external communication interfaces. The internal interfaces include two independent communication channels: a CAN bus interface for user-space communication and a RS-485 interface for reconfiguration purposes.

As external interfaces, a USB interface and a LAN interface are provided. The USB interface (FT232R) acts as a UART. The LAN interface is built on foundation of the WIZnet chip W5100, which is a hardwired TCP/IP embedded Ethernet controller. We have chosen this Ethernet controller to save program memory and CPU load on the main microcontroller. Another fact which distinguishes the Communication node from the other nodes is the different node logic controller (C_{comm}) which has two additional external interfaces for reconfiguration.

Furthermore there are different types of Execution nodes. So far, we developed a digital-analog I/O node and a motor driver node (Fig. 5). The I/O node has 16 digital outputs, 8 digital inputs and 8 analog inputs. The motor driver node is mainly based on the power motor driver VNH2SP30-E from STMicroelectronics. One specialty of the motor driver node is that its communication channels are fully optocoupled. This ensures that no electrical noise from a connected motor interferes with the communication or with the internal electrical system in general.

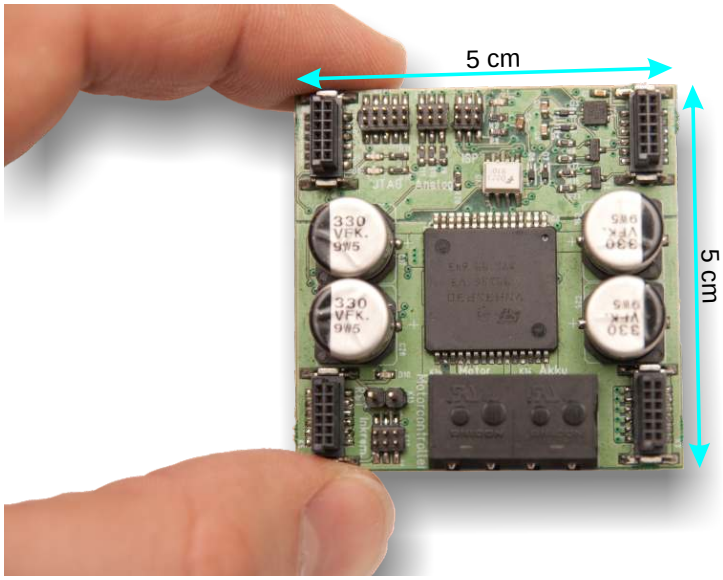


Fig. 5. One node of our stack system (motor driver node)

6 Related Work

The term reconfiguration in terms of embedded system often is related to Field Programmable Gate Arrays (FPGAs). There are many relations between an FPGA-based system and our modular multi-microcontroller system. In terms of an FPGA system coarse-granular reconfiguration means the replacement of complete system modules in contrast to only reconfiguring parts of a processor. Our reconfiguration is coarse-granular in terms of only reconfiguring a complete module. Additionally, we integrate I/O-hardware, power drivers and even galvanic isolation in our modules. Masselos and Voros [5] introduce a classification of reconfigurable architectures. Our approach cannot directly be classified by their classification scheme. We have a temporal computation style, have a great post fabrication programmability and are highly flexible. Our type of reconfiguration is kind of both static—from microcontroller's point of view—and dynamic—from the system's point of view.

Wahlah and Gossens [7] propose a 3-tier reconfiguration model for FPGAs using hardwired network on chip. Besides their 3-tier architecture they use the hardwired network as a dedicated communication channel for reconfiguration. They also propose a separation of concerns as the user has no need to bother about reconfiguration details as the application manager takes over this task.

Blodget et al. [1] present an approach for dynamic reconfiguration of a special FPGA. They propose a hardware and software infrastructure to enable the FPGA to reconfigure itself using a soft microprocessor to control the reconfiguration. We also use the concept of a dedicated microcontroller to control the

reconfiguration. In our case each module carries its own reconfiguration microcontroller which is also used for the automatic structure detection.

The technical term "component-based reconfiguration" is usually used for software systems but there are many related issues to a module-based system as we have proposed it in this paper. Matevska [6] "presents a model-based approach to runtime reconfiguration of component-based systems, which aims at minimising the interference caused by the reconfiguration and thus maximising system responsiveness during reconfiguration." His main goal is to maximize the system responsiveness during reconfiguration. In contrast, our main goal is to encapsulate the reconfiguration process to ease the reconfiguration. The infrastructure we proposed ensures that—except the modules that actually get reconfigured—the rest of the system remains operational.

Chen et al. [2] "propose a framework to support component-based model integration, hierarchical functionality composition, and reconfiguration of systems [...]". Their framework is more related to our future work but they also use hierarchical components to hide the implementation details. This is comparable with our separation of concerns paradigm which hides the details of reconfiguration.

David et al. [3] propose a multi-stage approach for reliable dynamic reconfiguration. They focus on a validation of the reconfiguration process to detect errors before the execution of the reconfiguration. This partly also applies to our infrastructure regarding the separation of concerns paradigm. We provide an interface to the user to reliably reconfigure a system's module. Another part of the work of David et al. is the error detection in a running system and to automatically mitigate them by reconfiguration. This part of their work is more related to our future work.

Gumzej et al. [4] propose a reconfiguration pattern for UML-based projects of embedded real-time systems. Their concept mainly regards real-time capability of the reconfiguration management. We have not analyzed the real-time capability of our infrastructure yet and in their point of view the infrastructure we proposed would only be a part of the reconfiguration management. Our infrastructure proposal would be located on the hardware level and on the middleware level.

7 Conclusion and Future Work

Our intention was to create an infrastructure for coarse-grain flexible run-time reconfiguration of multi-microcontroller systems. We have shown that the infrastructure we proposed fulfills the requirements of a flexible module-based coarse-grain run-time reconfiguration and moreover introduces a separation of concerns regarding user-space communication and reconfiguration. Our infrastructure is able to cope with structural system changes and ensures a reliable reconfiguration. Additionally, our infrastructure provides the ability to do a multicast reconfiguration of functionally identical modules.

Until now, we have tested the automatic structure detection within one stack, and we have implemented a driving unit. The driving unit includes three motor

controllers realized on three Execution nodes and one Communication node. We successfully tested both the user-space communication and the reconfiguration communication. We are currently working on the bootloader code.

This infrastructure will be the foundation of future work. Both ideas, to extend the amount of modules and to more deeply integrate the stack system with complex embedded systems, will be issues in our future work. More complex functional units including embedded PCs will be developed to build a self-monitoring and self-repairing subsystem that uses in-system runtime reconfiguration to make the whole system more robust against failure.

References

1. Blodget, B., McMillan, S., Lysaght, P.: A lightweight approach for embedded reconfiguration of fpgas. In: DATE '03: Proceedings of the conference on Design, Automation and Test in Europe. p. 10399. IEEE Computer Society, Washington, DC, USA (2003)
2. Chen, W., Xie, C., Shi, J.: A component-based model integrated framework for embedded software. In: Embedded Software and Systems. Lecture Notes in Computer Science, vol. 3605, pp. 563–569. Springer Berlin / Heidelberg (2005), http://www.springerlink.com/content/2xq8nwktc0cr7d2w/
3. David, P.C., Lger, M., Grall, H., Ledoux, T., Coupaye, T.: A multi-stage approach for reliable dynamic reconfigurations of component-based systems. In: Distributed Applications and Interoperable Systems. Lecture Notes in Computer Science, vol. 5053, pp. 106–111. Springer Berlin / Heidelberg (2008), http://www.springerlink.com/content/9j85j27363v88342/
4. Gumzej, R., Colnari, M., Halang, W.A.: A reconfiguration pattern for distributed embedded systems. Software and Systems Modeling 8(1), 145–161 (February 2009), http://www.springerlink.com/content/nr3t327414250784/
5. Masselos, K., , Voros, N.S.: Introduction to reconfigurable hardware. In: System Level Design of Reconfigurable Systems-on-Chip. pp. 15–26. Springer US (2005), http://www.springerlink.com/content/g76x276332511272/
6. Matevska, J.: Model-based runtime reconfiguration of component-based systems. In: WUP '09: Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010. pp. 33–36. ACM, New York, NY, USA (2009)
7. Wahlah, M., Goossens, K.: 3-tier reconfiguration model for fpgas using hardwired network on chip. In: Proceedings of the International Conference on Field-Programmable Technology (December 2009)