Design of Embedded and Cyber-Physical Systems Using a Cross-Level Microarchitectural Pattern of the Computational Process Organization

Vasiliy Pinkevich[a], Alexey Platunov[a] and Yaroslav Gorbachev[b]

[a]ITMO University, Kronverksky prospekt 49, Saint Petersburg, 197101, Russian Federation
[b]LMT Ltd., Birzhevaya liniya 16, Saint Petersburg, 199034, Russian Federation

Abstract

The complexity and low efficiency of the reuse of various computational mechanisms and conceptual solutions at the architectural and microarchitectural levels during the creation of embedded and cyber-physical systems is still an open problem. The paper proposes a microarchitectural pattern that allows to represent the structure of the system being designed at various levels of abstraction and to guide the design process methodologically. The concept of the kernel, a special functional block, is introduced, which acts as the central reusable element of the pattern. The pattern organization the principles of the computational process representation in it are based on the aspect design model, the multi-level representation of the embedded system, and abstractions of the computing mechanism, platform, virtual machine. An example of the application of the proposed approach in some projects of distributed heterogeneous embedded systems is presented.

Keywords

embedded system, cyber-physical system, microarchitecture, SLD, design space exploration, aspect-based design, kernel

1. Introduction

The need for mass design and programming of microprocessor-based controllers is determined by the further rapid digitalization of society. The era of cyber-physical systems (CPS) and the Internet of things presuppose deep automation of almost all areas of human life and activity [1]. The use of a limited number of standard solutions (info-communication design platforms) for a variety of different tasks, without being able to influence the internal, deep structure of these platforms, significantly limits the developer's capabilities and degrades the quality of the product. However, existing custom low-level design and programming technologies for embedded systems (ES) and networked (distributed) embedded systems are still very troublesome and time-consuming. The results of such design are poorly scalable and have a low reuse ratio [1, 2]. Moreover, a paradoxical situation arises when technologies for creating embedded and cyber-physical systems aimed at implementing one of the key propositions of the Industry 4.0

Proceedings of the 12th Majorov International Conference on Software Engineering and Computer Systems, December 10–11, 2020, Online & Saint Petersburg, Russia

vupinkevich@itmo.ru (V. Pinkevich); aeplatunov@itmo.ru (A. Platunov); yaroslav-go@yandex.ru (Y. Gorbachev)

0000-0002-8635-5026 (V. Pinkevich); 0000-0003-3003-3949 (A. Platunov); 0000-0001-5419-6422 (Y. Gorbachev)

initiative, namely, creating a fully automated production of customized goods "using a unified production line", exclude automation systems themselves from the list of such custom systems.

One of the ways to solve the problem is to improve the technologies for creating complex embedded systems in the direction of developing methods and tools of the microarchitectural level in the end-to-end design route [3, 4]. In this case, the microarchitectural representation of an embedded system or its part (for example, a controller) can be defined as the most important meta-level in the hierarchy of the design route abstractions [3, 5, 6, 7].

Here we use the term microarchitecture as the organization of a system or its component at a level below the level of the architectural description (the responsibility of the architect), but above the level of the final implementation (software code, hardware schematics, etc.). In fact, these are the abstractions and principles of organization that the developer uses when creating an implementation of the architecture defined in the specification.

This information in many projects (usually low-budget, although quite complex) remains poorly documented (as good as comments in the source code). This leads to an excessively large gap between the architectural specifications and the implementation. Because of this, the implementation becomes non-transparent, and the system design is difficult to verify, maintain, and upgrade. As a result, it can become unmanageable.

Complex embedded systems projects in most cases, along with behavioral (functional) requirements, are characterized by many so-called non-functional requirements. It is useful to divide them into the final product requirements (for example, performance, reliability, energy efficiency) and requirements related to process/phases of the product life cycle, such as design, testing/certification, replication, maintenance, modernization, etc. The complex nature of ES design, which requires the developer to work with the requirements in a balanced manner, is still poorly supported with industrial-level methods and tools. The above examples of non-functional requirements in hardware and software projects are most often implemented and controlled as a residual. The situation is even worse with the technologies and tools for CPS creating, since requirements from the application area related to the automation object are added to the list of own ES design requirements [8, 9].

A large number of research groups are working to overcome these problems, the results of which are methodologies and tools related to the areas of "HW/SW Codesign" [10], ESLD [6, 11], aspect-based design [12, 13], languages for architectural design (architecture description languages, ADL) [14], which are based on the advances in systems engineering [9, 15, 16].

This paper proposes a set of formalisms aimed at organizing and managing general and application-specific (including local) design routes with an emphasis on the project stages preceding the implementation phase (architectural and microarchitectural design phases). Formalisms allow us to have a holistic view of the architecture of the product being created and its transformation while moving along the design route, as well as actively use and, if necessary, extract reusable design artifacts of various abstraction levels.

2. Problem statement

Existing reuse strategies during ES creation do not provide an opportunity to find an acceptable compromise between the design and development efforts and the quality of the product

being created. As a result, to obtain the optimal result from an engineering point of view, it is necessary to develop the system essentially from scratch. On the other hand, if it is necessary to reduce the development costs, a complete solution is used without the possibility of proper adaptation to the project requirements. This happens because the traditionally used design platforms are characterized by at least one of the following disadvantages:

- insufficient documentation of engineering solutions (only final implementation is available);
- weak possibilities for configuring (adapting) the proposed engineering solutions to achieve the required characteristics;
- incomplete coverage of the stack of the necessary ES organization level (only the software level or only the level of the hardware platform);
- incomplete coverage of the necessary aspects (requirements) of the project.

For example, real-time operating systems (FreeRTOS, eCos, Embedded Linux) represent only the level of the system software, and often do not include components for implementing such important service functions as in-system software updates. However, at the same time, they may have ports for different families of microcontrollers and include rich standard driver libraries.

The ready-made programmable logic controller (PLC) covers the full stack of levels from hardware to user programming but is used as a "black box" without the possibility of any significant adaptation of its platform to the requirements of the project (adding new service functions, optimization to reduce costs).

A family of microcontrollers with a library of standard drivers (for example, the STM32 family and the STM32Cube tools) partially cover the ES hardware and system software layers, but the rest must be developed from scratch. Moreover, even the use of ready-made driver implementations is often problematic due to their inflexibility and low level of optimization by performance, memory consumption, etc.

We see the possibility of resolving the existing situation in the use of special design patterns as reusable objects, that cover the full set of organization levels [17, 18] and design aspects [19] for the target class of computing systems, as well as including several levels of description of engineering solutions. We will call such objects microarchitectural design patterns. This highlights the critical importance of the microarchitectural level of the project documentation for the project maintenance.

3. Aspect-based design

The complex nature of ES projects, combined with their growing complexity, requires the creation of design methods and technologies that will effectively consider, analyze, synthesize and track the quality of all recognized as essential parts of the ES organization and the infrastructure existing around it throughout the life cycle, especially at the stages of creation and modifications. Isolation of such relatively independent parts is a nontrivial process. We will

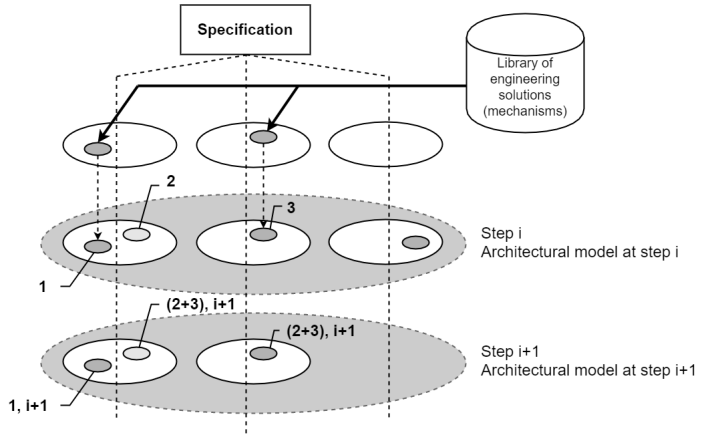

Figure 1: Steps of the aspect-based design process

refer to such localized parts of a project or target system as aspects. In other words, aspect is some particular design problem within the ES creation problem. Let us emphasize once again that aspects do exist not within the framework of any stage or step in the development of a project or target system, but they exist throughout the entire design process or the entire life cycle of the system (the "weight" of an aspect in a project changes over time and can degenerate to zero). The set that includes all aspects of the design will be called the aspect space of the ES design process. The set that directly belongs to the target system under creation will be called the aspect space of the target system [19].

So, an aspect is an artificially allocated segment of the design space, reflecting a particular problem of the project during its implementation (conceptual, local aspect). The designer himself forms a list of aspects, which he then uses. The designer highlights the conceptual aspects that exist throughout the whole project duration and can highlight, if necessary, local aspects at individual steps of the project (see Fig. 1) [19, 20].

Fig. 2 shows the result of transforming a traditional Y-diagram of a computing system into an aspect diagram of an ES project:

- the center of the diagram, depending on the chosen interpretation of concentric circles, is the endpoint of the design process (implemented by ES) or the starting point (initial specifications);
- concentric circles indicate the levels of the target system hierarchy or steps in the design process;
- axes correspond to the allocated (conceptual and local) aspects of the project;
- local aspects can cover a limited number of levels or project steps;
- points at the intersections of axes and circles are significant categories of objects or processes of the target system or ES project [21, 22].

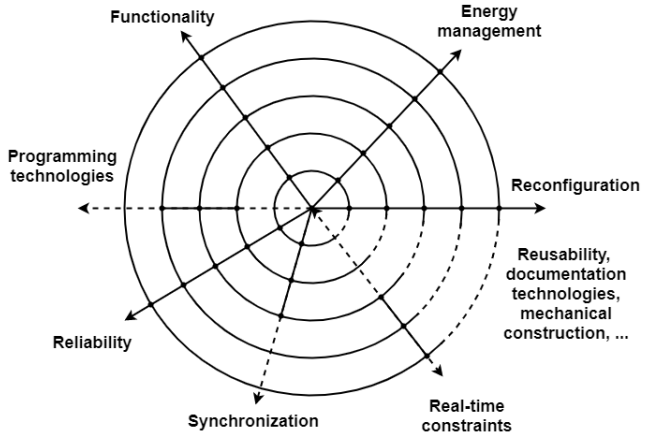

Figure 2: Aspect diagram of a project

The design process for complex ES should involve a hierarchical representation of computational platforms, functional description languages with their translators, and application-level solutions in a unified system of abstractions. This representation is aimed at project (functional and non-functional) aspects description, management, and maintenance in a uniform style, which makes it possible to increase the unification in the design of all ES components. The central idea is the consistent refinement/elaboration of the target system through the hierarchy of projects with a decreasing level of abstraction.

Fig. 3 illustrates an ES design process pattern based on the aspect model and composition of computational platforms. Its advantages are:

- parallel balanced work with functional and non-functional requirements;
- generation of architecture from the basis of the proposed system of abstractions;
- combining the design and execution phases of the computational process within a single space of engineering solutions;
- a deferred phase of hardware-software partitioning.

4. Microarchitectural pattern

In any relatively complex computing system, the project designer can extract patterns that allow some variation in implementation within the boundaries of this project. This is important both for the successful fulfillment of the requirements of the technical assignment and for improving the quality of the project itself due to:

- providing a reserve of flexibility to fix bugs;
- the ability to easily adapt to new requirements arising during validation, integration, etc.;

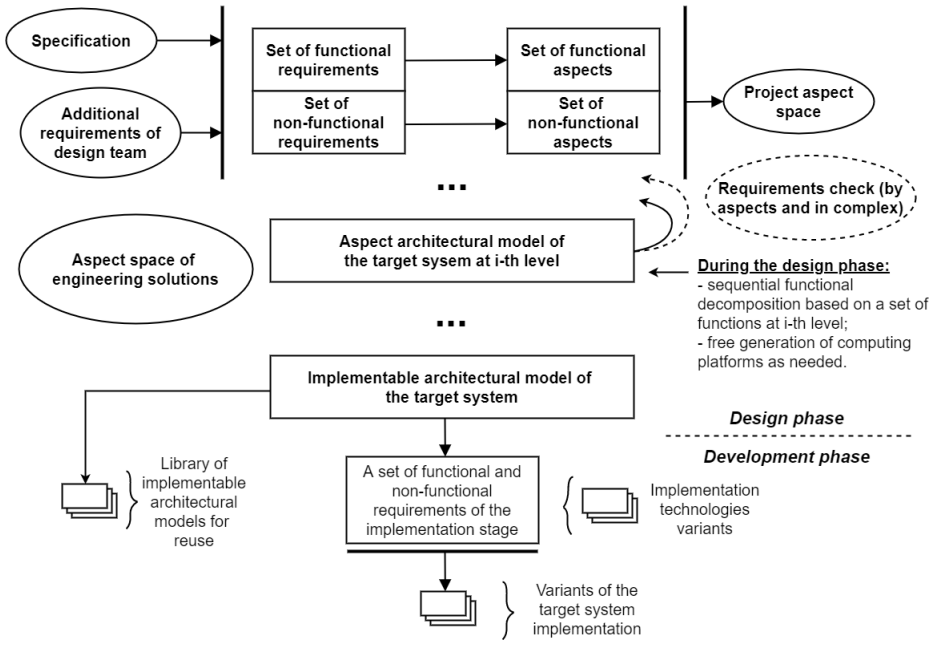

Figure 3: ES design flow based on the aspect model and composition of computational platforms

- providing the potential for modernization;
- providing the possibility to reuse engineering solutions.

The microarchitectural pattern is designed to create complete systems of a certain class, so it is an object of a coarse granularity and has a fairly narrow scope of application. It is a pattern for both the target system and its design process.

The microarchitectural pattern differs from the framework, library, and platform in its coarser granularity and self-sufficiency, as well as in the mandatory availability of information at the architectural and microarchitectural levels, although in a particular case it can be reduced to them at the implementation level.

The microarchitectural pattern differs from design patterns both in its coarser granularity and specialization, and in its deeper implementation, and there may be several options for implementing the pattern, both at the microarchitectural level and at the final implementation level.

Let us consider the complex microarchitectural pattern applied to the design of the distributed automation systems controllers implemented on a microcontroller platform. Such a platform includes one or more programmable microprocessors (including homo- and heterogeneous multicore ones) with a set of coprocessors, input-output controllers, and memory devices (both included in the microcontroller and external).

Since the microarchitectural pattern corresponds to a complete system, part of the pattern is a subsystem focused on some particular task. Like the system as a whole, such a subsystem cannot be implemented using any standard library component, but is a specialized composition

of ready-made and original technical solutions. In the context of a microarchitectural pattern, we will call such a subsystem a kernel. The kernel is a specially allocated functional unit that has the following properties:

- clearly defined functionality;
- several possible implementation variants at the software, hardware, or software-hardware level;
- vectors of characteristics in various design aspects;
- the ability to scale by functionality and required resources;
- significant potential for reuse in a certain class of projects.

During the design process of distributed automation controllers, it is convenient to represent as kernels the following elements:

- components responsible for input/output and networking;
- basic mechanisms for controlling the computational process (timers, interrupt system, scheduler, etc.);
- service functions (diagnostics, remote and in-system updating, configuration, information security, etc.);
- standard data processing functions (encryption algorithms, work with various data/file formats, filtering, image processing, etc.);
- virtual machines, translators.

One of the most important properties of the kernel, in contrast to other functional blocks, is its focus for reuse. Firstly, its microarchitectural specification, and then its final implementations. Kernel reuse is possible in the following ways (see Fig. 4a):

1. At the implementation level (source codes and hardware are reused).
2. At the microarchitecture level (the way of computations organization and hardware/software partitioning (if any) is reused).
3. At the level of the functional interface (only the functionality and methods of interfacing with components of the same level are reused).

Design and development of kernels require the biggest part of work when implementing a project “from scratch”, but the requirements for them are usually weakly connected with the applied problem being solved and are often not detailed in the system specification. They are primarily responsible for organizing system and service processes. At the same time, the main (non-functional) applied characteristics and properties of the target system (cost, performance, power consumption, reliability, maintainability, etc.) depend on the quality of implementation of very these objects. In this regard, the set of kernels is the backbone for the controller project, and their design and implementation are the most critical parts of the project.

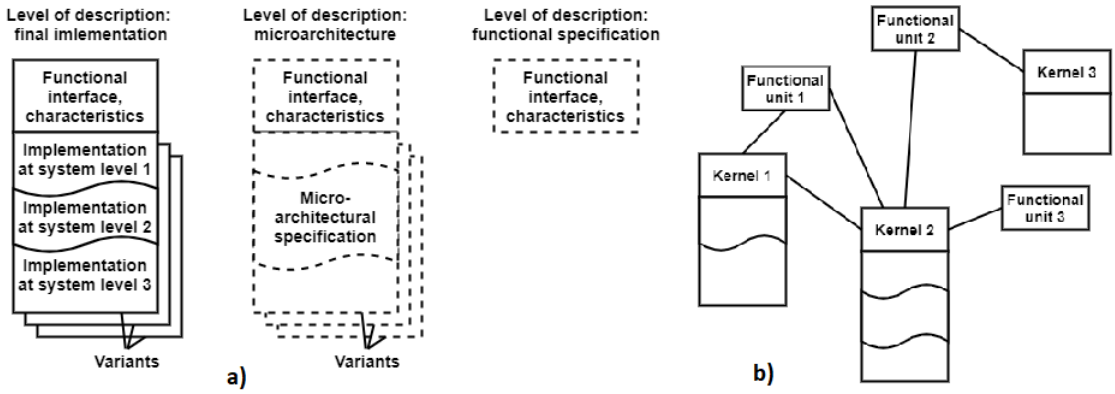

Figure 4: a) Levels of description and reuse of the kernel. b) An example of combining kernels within some organization level of the system

Since kernels can use lower-level kernels, their functionality and interface should be designed as versatile and flexible as possible, while keeping the overhead low. In this regard, the allocation of kernels may turn out to be nontrivial and not interfere with the usual functionality of drivers in existing systems.

The vector of kernel characteristics is determined both by the its functions and by the variant of its implementation, including the characteristics of the lower-level kernels used.

The basis of the microarchitectural pattern is the integration of kernels in a single scalable structure. Since each kernel can have several implementation options, the pattern is flexible and adaptable to the requirements of a particular project (see Fig. 4b).

During the design process using a specific pattern, first of all, the required configuration and the necessary expansion of the pattern are defined. This provides conditions for the implementation of application functions with a given quality and the fulfillment of non-functional requirements. Then, on this basis, the main functionality of the system is developed.

From the target system design process point of view, the microarchitectural pattern solves the following tasks:

- specifies a set of requirements for the internal system organization (they are usually absent in the initial specification);
- offers a set of engineering solutions for the target functions implementation;
- provides methodology on the use of engineering solutions;
- defines the segment of the design space in which the target system will be created;
- adjusts the design process by refining aspect weights and identifying local aspects.

Setting target characteristics for the pattern as a whole and each kernel imposes restrictions on the choice of implementation options for each kernel in the pattern, including the pattern microarchitecture itself. Due to the many options for combinations of characteristics,

the choice of a particular option is a non-trivial task. Provided a kernel characteristics formalization exists, mathematical optimization methods are applicable (annealing method, genetic algorithms, etc., used, i.e. in high-level synthesis tools [10, 23]).

Certain sets of requirements may not have appropriate implementation options within the existing pattern and the pattern will need to be extended.

Depending on the degree of pattern kernels reuse, the following scenarios for using a pattern in the design process are possible:

- direct use, only proper configuration is required;
- partial implementation changes (for example, because of a partial change in hardware components);
- partial redesign including developing new kernels and microarchitecture changes (for example, if there are specific requirements or significant changes in hardware components).

A complete redesign of the pattern should only be required when the requirements or class of the system change radically.

Each new project that required a change in the microarchitectural pattern enriches it with new engineering solutions. This expands the pattern and therefore increases its value as a reusable artifact. To achieve this effect, all engineering solutions in the pattern must be properly designed and documented, the necessary usage methodology must be provided. Thus, the method of describing a pattern can be considered as a type of architectural style or architecture description language [24, 25], focused on displaying the relationships between key objects and groups of reusable objects in the project.

Specification of a microarchitectural pattern consists of the following elements:

1. Kernels specifications:
    - description of functionality, dependencies, characteristics;
    - microarchitectural specifications of different implementation variants;
    - reusable implementations (including the necessary configuration and automation tools).
2. Guidelines for:
    - use of kernels and their integration;
    - expansion of kernels with new variants of microarchitectural and target (low-level) implementation;
    - development of new kernels.

5. Experiments and results

As an example of the application of the proposed approach, let us consider a pattern developed and used at LMT Ltd. [26] to create controllers for various purposes. The pattern covers

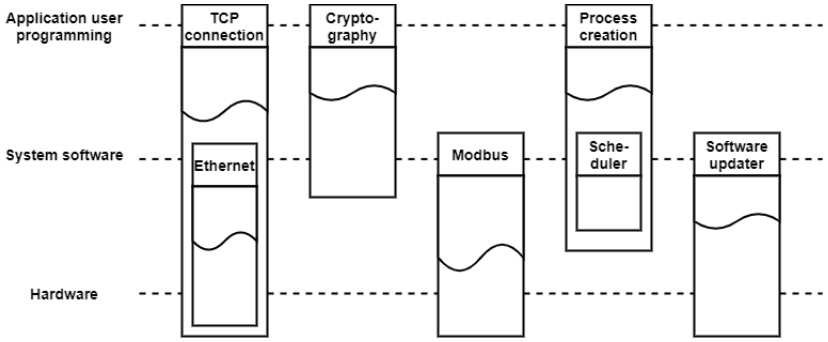

Figure 5: Examples of kernels in the pattern

three levels of controller organization (hardware, system software, and application software) to provide complete coverage of design tasks. The pattern uses: microcontrollers of the Coldfire family; a typical set of circuitry solutions and electronic component base; in-house system and application software platforms.

The pattern includes the following kernels:

- low-level input-output drivers (including UART, CAN, I2C, SPI, Ethernet);
- drivers for various protocols of controller networks and the Internet (including Modbus, CANPro, TCP/IP, in-house secure Internet protocol);
- drivers for supporting various memory chips and specialized devices (including DataFlash, GSM modems, radio transceivers);
- mechanisms of organizing parallel processes and interprocess communication;
- service mechanisms (diagnostics, in-system software update);
- a virtual machine of the user application programming language and the library of standard functions.

Depending on their functions, kernels cover one to three levels of organization (see Fig. 5).

Let us list and briefly describe some projects implemented using this pattern:

1. A PLC for distributed automation systems SCG-4. The controller uses all three levels of the organization presented in the pattern. A project-specific extension of the pattern is the external input/output and synchronization controllers implemented on the FPGA connected to the microcontroller.

2. Network communicator of SCG-3.6 family for automated city lighting control system and automated energy measurement system. The controllers do not use the user programming layer because they have a fixed mission and do not imply changes in functionality by the end user. They most widely use the set of communication channels and protocols supported by the pattern.

3. Controller of the scanning probe microscope MiniLab. The main mission of this controller is the implementation of high-performance digital signal processing to control measuring equipment. It is based on two patterns:

- controller of automation systems used as a control and communication core;
- FPGA-based digital signal processing processor with its own specific set of input-output controllers.

The second pattern also implements a three-level organization of the computational process and has its own level of user application programming [27].

Due to the high level of engineering solutions unification within the pattern, all kernel implementations are reused in the listed and other projects, which significantly (up to 50-80 %) reduces efforts required for creating a new version of controller.

It is planned to port the pattern to a new family of microcontrollers, which will require to create new implementations of all kernels that directly interact with the functional blocks of the microcontroller. However, due to the reuse of functional interfaces and microarchitectural specifications, it will be possible to carry out practically seamless transfer of the applied functionality implementations and application-level user software.

Thus, the proposed approach to the allocation and structuring of reuseable objects in CPS and ES projects based on microarchitectural patterns demonstrates high efficiency when creating several systems of similar purpose and/or long-term maintenance of the single system.

In contrast to the existing solutions, the proposed approach involves the conscious formation and reuse of variable patterns at the level of the entire system, instead of developing similar systems almost "from scratch" on the basis of fine-granularity reusable objects.

Fig. 6 shows an example of a microarchitectural description of the industrial automation controller project with focus of the infrastructure for remote data storing to the FLASH memory area of application programs. This subsystem is used when updating the user image of the software and recording settings (only the part of the process that is directly responsible for data transfer is displayed).

The figure contains elements of the computational process connected by arrows that show a logical end-to-end sequence of actions (but do not display secondary transitions, loops, and other feedbacks). The boxes below the description of the actions show the microarchitectural elements and mechanisms used at each stage (implemented both at the purely software and hardware levels, and with a mixed implementation).

6. Conclusion

Improving the design efficiency of embedded and cyber-physical systems is a very complex and time-consuming task, which today is solved with low-level patterns, prototypes, and the implementation-level reusable elements. This narrows the search space for design solutions, complicates scalability, and reduces the effectiveness of verification, debugging and testing tools. At the same time, “direct” attempts to implement complex projects at an abstract level, followed by hardware-software partitioning, are “stalling”.

We propose a compromise option that increases the abstraction degree of reusable (documentable) solutions, methodologically “pushing” developers to this design style.

The question of choosing/creating a convenient and effective expressive means for this kind of abstractions with access to the automatic generation of an implementation-level spectrum

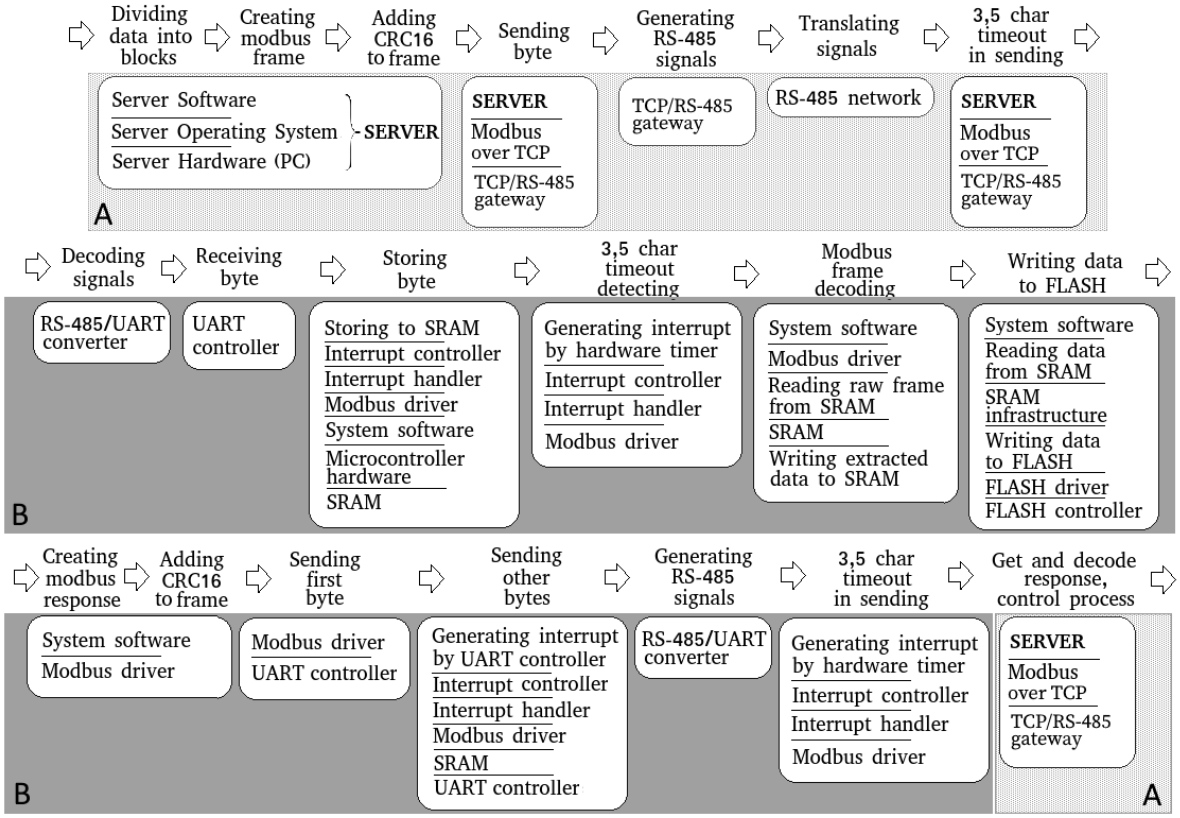

Figure 6: Example of a microarchitectural description: infrastructure for remote data storing to the controller's FLASH memory. A - microarchitectural elements of the external infrastructure, B - microarchitectural elements of the controller.

of solutions remains open.

References

[1] M. Masin et al., "Cross-layer design of reconfigurable cyber-physical systems," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, 2017, pp. 740-745, doi: 10.23919/DATE.2017.7927088.

[2] M. Khalil, "Improving Solution Reuse in Automotive Embedded Applications using a Pattern Library Based Approach," 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Munich, Germany, 2019, pp. 653-659, doi: 10.1109/MODELS-C.2019.00100.

[3] D. Densmore, R. Passerone, A. Sangiovanni-Vincentelli, "A Platform-Based Taxonomy for ESL Design", IEEE Design and Test of Computers, September 2006.

[4] Edward A. Lee. 2018. Modeling in engineering and science. Commun. ACM 62, 1 (January 2019), 35–36. DOI: https://doi.org/10.1145/3231590

[5] Sangiovanni-Vincentelli A., Damm W., Passerone R. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. Eur. J. Control. 2012. Vol. 18. P. 217—238.
[6] B. Bailey, G. Martin, “ESL models and their application”, New York: Springer Publication, 2010.
[7] 7. Panagopoulos, G. Papakonstantinou, N. Alexandridis, and T. El-Ghazawi, “A comparative evaluation of models and specification languages for Embedded System design”, Languages, Compilers, and Tools for Embedded Systems (LCTES-03), San Diego, Ca., June 11-13, 2003.
[8] D. Broman, Ed. A. Lee, S. Tripakis, and M. Toerngren, “Viewpoints, formalisms, languages, and tools for cyber-physical systems”, 6th International Workshop on Multi-Paradigm Modeling - MPM’12, October 2012, pp.49–54.
[9] S. Bonnet, F. Lestideau, J.-L. Voirin. Arcadia and Capella on the Field: Real-World MBSE Use Cases. MBSE Symposium, Canberra, October 27th, 2014.
[10] 10. J. Teich, “Hardware/software codesign: the past, the present, and predicting the future”, Proceedings of the IEEE, 2012, vol. 100, pp.1411 – 1430.
[11] A. Sangiovanni-Vincentelli, “Quo vadis SLD: reasoning about trends and challenges of system-level design”, Proceedings of the IEEE, 95(3), 2007, pp.467-506.
[12] J. M. P. Cardoso, P. C. Diniz, J. G. de F. Coutinho, and Z. M. Petrov, “Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach (Google eBook)”, Springer, 2013, p. 215.
[13] A. Platunov, and A. Nickolaenkov, “Aspects in the design of software-intensive systems”, 2012 Mediterranean Conference on Embedded Computing (MECO), June 2012, pp.84-87.
[14] P. H. Feiler, D. P. Gluch, and J. J. Hudak, The Architecture Analysis & Design Language (AADL): An Introduction. Software Engineering Institute, 2006.
[15] “ISO/IEC/IEEE 42010 Systems and software engineering — Architecture description.” 2011.
[16] Essence – Kernel and Language for Software Engineering Methods. Foundation for the Agile Creation and Enactment of Software Engineering Methods (FACESEM) RFP, 2012.
[17] Kluchev A., Platunov A., Penskoi A. HLD methodology in embedded systems design with a multilevel reconfiguration. Proceedings - 2014 3rd Mediterranean Conference on Embedded Computing, MECO 2014 - Including ECyPS 2014, 2014, pp. 36-39
[18] Pinkevich V., Platunov A. Method for Testing and Debugging Flow Formal Specification in Full-Stack Embedded Systems Designs. 9th Mediterranean Conference on Embedded Computing, MECO 2020 - Proceedings - 2020, pp. 9134213
[19] A. Platunov, A. Kluchev, and A. Penskoi, “HLD Methodology: The Role of Architectural Abstractions in Embedded Systems Design”, 14th GeoConference on Informatics, Geoinformatics and Remote Sensing, 2014, pp. 209–218.
[20] Penskoi A., Platunov A., Andreev V. The Selection Problem and Evaluating Method for Architectural Design Tools of Embedded Systems. 8th Mediterranean Conference on Embedded Computing, MECO 2019 - Proceedings, 2019, pp. 150-154
[21] A. Chattopadhyay, “Ingredients of adaptability: a survey of reconfigurable processors”, VLSI Design, 2013, vol. 2013, p.18.
[22] Pinkevich V., Platunov A. Using architectural abstractions in embedded system design. Proceedings of the 4th Mediterranean Conference on Embedded Computing (MECO 2015),

Works in Progress in Embedded Computing (CD ROM) - 2015, Vol. 1, No. 1, pp. 3-6
[23] Platunov A. E., Yanalov R. I. Design of computing platform for cyber-physical systems. Journal of Instrument Engineering. 2017. Vol. 60, N 10. P. 993—998 (in Russian).
[24] Clements P.C. A survey of architecture description languages. Proc. 8th Int. Work. Softw. Specif. Des. – 1996. – pp. 16–25.
[25] Shaw M. Comparing architectural design styles. IEEE Softw. – 1995. – V. 12 – No. 6 – pp. 27–41.
[26] LMT Ltd. URL: https://lmt.spb.ru (accessed 01.11.2020)
[27] Pinkevich V.Y., Platunov A.E., Penskoi A.V. The Approach to Design of Problem-Oriented Reconfigurable Hardware Computational Units. Wave Electronics and its Application in Information and Telecommunication Systems (WECONF 2020) - 2020, pp. 9131512