A (Re)Configuration Mechanism for Resource-Constrained Embedded Systems

Guo, Yu; Sierszecki, Krzysztof; Angelov, Christo K.

Published in:
Proceedings of 32nd Annual IEEE International Computer Software and Applications Conference

Publication date:
2008

Document version
Final published version

Citation for pulished version (APA):
Guo, Y., Sierszecki, K., & Angelov, C. K. (2008). A (Re)Configuration Mechanism for Resource-Constrained Embedded Systems. In *Proceedings of 32nd Annual IEEE International Computer Software and Applications Conference* (pp. 1315-1320). IEEE Computer Society Press.

A (Re)Configuration Mechanism for Resource-Constrained Embedded Systems

Yu Guo, Krzysztof Sierszecki and Christo Angelov
Mads Clausen Institute for Product Innovation, University of Southern Denmark
Alsion 2, 6400 Soenderborg, Denmark
{guo, ksi, angelov}@mci.sdu.dk

Abstract

An embedded application, developed under the COMDES framework, is built from reusable components implemented as executable function blocks. These are stored in a component repository in a relocatable binary format, whereby the application is configured from prefabricated components, rather than generated. This paper presents a configuration mechanism and a tool supporting the process of configuration, which makes it possible to automatically generate application binaries out of COMDES application specifications. Moreover, the obtained code must not be entirely replaced in case of possible updates, i.e. it is reconfigured such that only a part of the code needs to be modified with a new configuration.

1 Introduction

COMDES (Component-Based Design of Software for Distributed Embedded Systems) is a software framework [1], which has been developed in an attempt to address the design issues of distributed embedded systems, with a significant attention to the requirements of resource-constrained applications.

The COMDES generative development process [2], involving modeling of embedded applications and automatic synthesis of application codes, can be conceptually illustrated as in Figure 1.

A system is designed in its application domain at a relatively high level, e.g. a control system designed in a MATLAB/Simulink environment. Next, the domain model that satisfies application requirements is transformed into a COMDES model, e.g. by automatic mapping of Simulink components to COMDES components (under investigation). The obtained software model is further supplemented with information that guides the implementation generation. Ultimately, the synthesized code is deployed into embedded devices and tested against the real environment.

The implementation generation can be achieved in two ways: conventional program generation and system configuration. The former is characterized by the compilation and generation of new executable code, which has to be subsequently downloaded into the target system. Hence, it does not provide adequate support for software reconfiguration. However, embedded systems must be able to adapt to changes in requirements and facilitate the removal of software errors. The importance of system reconfiguration has been further emphasized by the latest trends in the area of embedded systems: real-time reconfiguration with zero downtime [3].

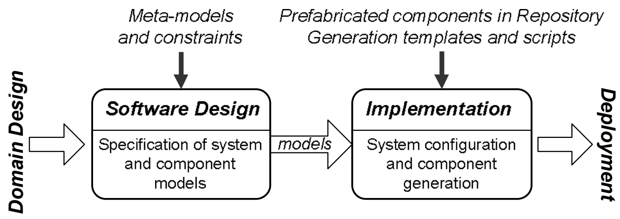
Meta-models and constraints
Prefabricated components in Repository
Generation templates and scripts
Domain Design
Software Design
Specification of system and component models
models
Implementation
System configuration and component generation
Deployment

Figure 1. The engineering approach to COMDES system configuration

The second approach follows a development line that started with industrial automation systems a few decades ago. In this case, the application is built from prefabricated executable components stored in non-volatile memory in binary format. Consequently, model-based configuration is substituted for model-based program generation, whereby the configuration specification is stored in data structures containing relevant information such as component parameters, input/output links, sequence of execution, etc. With this type of system, program generation is used in a limited manner in order to generate code for low-level components such as basic function blocks, to be included in the component library.

This approach is better suited for development of embedded software because it provides support for system reconfiguration, which is achieved by updating data structures, whereas executable codes remain unchanged. Therefore it has been adopted in the COMDES software development process.

Software reconfiguration can be implemented off-line, i.e. before the system is restarted or eventually – on-line, during system operation. However, on-line reconfiguration is a complex problem that has to be treated with caution because of the possible impact on system dynamics in terms of system stability, bumpless transfer of control during mode changes, etc. Some of the existing IEC 61131 programming systems support on-line updates. However, these systems are subject to various restrictions (e.g. only data can be updated), and require the support of a special run-time environment.

COMDES re-configuration does not address the entire problem of system reconfiguration but provides a basic technique for software reconfiguration. This technique makes it possible to carry out software reconfiguration by deploying only the modified executable code into a system, i.e. by applying a patch. Generation of the reconfigured executable code takes into account the currently running code stored in the embedded device, so the changes are minimal.

A tool called *Configurator* has been developed to support the process of configuration and reconfiguration. It reads the application specification (model) created during the specification stage, and produces the full configuration of the application and the corresponding codes to build the final application executable. In order to achieve this, the configuration process basically consists of two stages: *component selection*, which operates on the software model level, and *component assembly*, which deals with the binary level.

The rest of the paper is structured as follows: Section 2 introduces the concept of executable component in the COMDES framework – the function block. Section 3 presents the first stage of the configuration process – component selection. The second stage – component assembly – is discussed in Section 4. A summary of the presented (re)configuration method is given in the concluding section of the paper.

2 Executable component

An executable component model is transformed into a piece of executable code – a deployment unit that could be loaded into a system to extend its functionality. The format of the executable code can be quite different ranging from source code to native binary code, depending on the system run-time environment.

Figure 3 presents three typical situations, where executable code is achieved by:

- interpretation of source code - typical for scripting languages,
- interpretation of byte code (intermediate form) - popular because of improved performance against interpretation of source code,
- direct execution of linked object code (native machine form) - the most commonly used method providing the best performance.

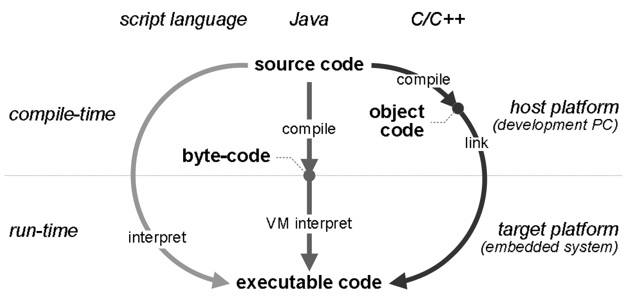

Figure 3. Transformation of source code into executable code

As pointed out in [4], choosing the form of executable code is a trade-off between putting a burden on assembling (compiling, linking) at compile-time and execution (interpretation) at run-time. Embedded systems are usually very limited in operational resources, e.g. memory, processor capacity, and therefore, it is natural to move the burden from a target device to a host computer. Obviously, the best option is to let the target device execute the native machine code directly without any interpretation, and to assemble the code at the host computer (see Figure 3).

In the COMDES framework, the smallest piece of executable code, the deployment unit, is a function block. Function blocks are used to configure higher-level components (actors) and applications [1].

The function block (FB) is the main COMDES component, which defines the standard structure and interface of an executable component, graphically shown in Figure 4.

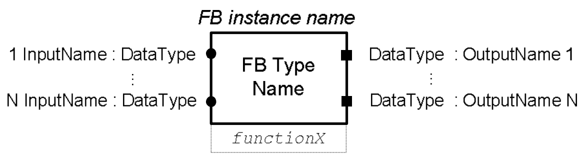

Figure 4. Function block

The FB must have at least one function, as well as a set of inputs and outputs, which accept and generate signals of a given data type. The type reflects representation of data that a Function Block is capable of processing – it can be a primitive or composite type, e.g. integer, float, vector. Furthermore, the FB definition specifies common information attributed to all function blocks, which is necessary for providing component reuse. The information is stored in a repository, and it can be later retrieved when the component is selected for an assembly.

COMDES defines five kinds of function block: basic, composite, signal driver, modal and state machine. A basic function block (BFB) is specified by subsets of inputs and outputs, as well as a number of functions transforming input signals into output signals. A function block diagram that specifies a set of FB instances softwired into a software circuit may be encapsulated into a composite function block (CFB), whereby constituent FB instances are executed in a sequence that is determined by the flow of signals in the diagram, see Figure 5.

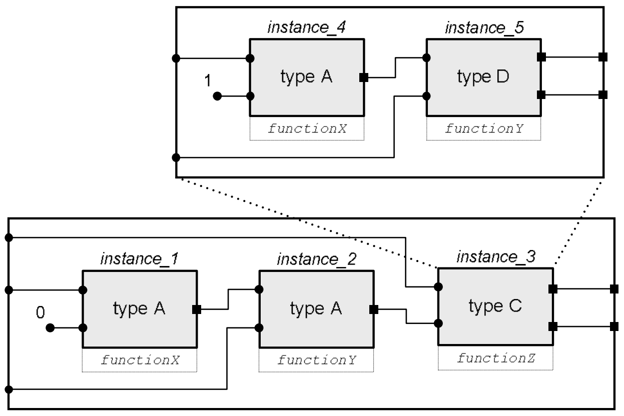

Figure 5. Composite Function Block built out of several Basic FBs

From an implementation point of view, a function block consists of a data structure containing all necessary information such as parametric and signal inputs, signal outputs and persistent variables, which is instantiated for each object of a given FB type and stored in RAM.

FB type defines the behavior of a function block determined by functions stored in ROM. FB instances of the same FB type share the same behavior and differ by the data structure. Typically, there are many instances of a function block of a given type.

FB is an executable component since its functions, written by skilled software engineers, are implemented for a number of different architectures/platforms and stored in a component repository (Figure 6). In contrast, an application is configured by domain experts from the already available function blocks found in the repository and code is not generated in the configuration process.

In order to match the limited resources of embedded systems, the COMDES framework is implemented in the C programming language.

As already mentioned, FB functions are not hardware-dependent and are compiled into relocatable object codes for a particular CPU architecture, e.g. AVR, H8/300. Some parts are dedicated to a particular hardware platform (e.g. hardware I/O drivers) and are written by an expert, once. In this way, portability and native platform performance are achieved rather easily, assuming existence of the tool-chain for the platform of interest, see Figure 6.

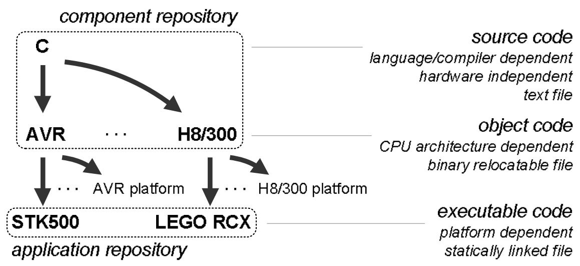

Figure 6. C source code as a portable code

A configuration is defined as a set of parameterized instances of function blocks and connections between them, in accordance with application requirements. The configuration process involves finding function blocks used in the application model in the component repository and assembling them to form a final executable that can be executed on the defined platform. In case of an update of the application, a reconfiguration process is applied, which allows the application to be partially changed by adding, removing and updating of components. To be specific, in case of updated configuration, only changed components will be assembled into the final executable, whereas the unchanged part will remain intact in the system.

3 Component selection

As soon as a new component is added into the system, it has an impact on a part of the system. The new component may refer to certain components, and also be used by other components. The situation will be the same if an existent component is deleted from the system. Hence, managing component dependencies is an important and complex issue in component-based application.

When an application developer uses the specification tool to design a COMDES application by selecting function blocks from the repository, there might be some components, such as composite function block instances, that depend on other function blocks stored in the repository. The developer should focus on whether or not the component can satisfy the application requirements, without concern for the dependency of function blocks. There should be a mechanism to help him with finding all the necessary components from the repository automatically, to generate the final configuration of the application. The final application configuration comes from the application specification and its dependency components. Once all dependencies of the application are detected, the configuration containing all instances and all function block types used in the application can be generated.

The dependency between components can be represented by a linked list (or matrix, see [5]). Each component specification has a description of dependency, specifying which other components are needed. Solving the dependency is actually a recursive searching routine. The search starts from the application specification, which is the top level of all components, and it will not stop until it reaches basic function blocks that do not depend on any other function block, as illustrated in Figure 7, where the symbol C_X stands for a CFB, whereas B_X for a BFB.

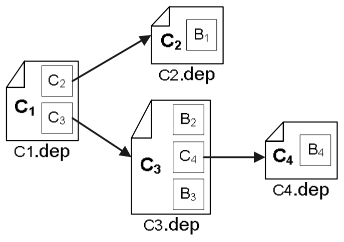

Figure 7. Dependency linked list

The linked list can be implemented as a file system where each node representing a component is a DEP file, as shown in Figure 7. The connection among nodes is represented as a path of the file system.

While components are being added into the application during the dependency resolving stage, they should be named appropriately and uniquely. The names of instances in the application model are specified by the developer. Since he has no knowledge about the instances inside a component instance, those internal instances should be named automatically. Provided that each two components are not allowed to have identical names, the rule of naming could be simply defined as *parent instance name + child instance name*, where the parent instance name is the one specified by the developer and the child instance name is the internal instance name that is shipped with the component.

To summarize, the first stage in the configuration is selection of components; the components in the repository should be added into the application model based on the dependency, in order to generate the configuration of the system with all components, see Figure 8.

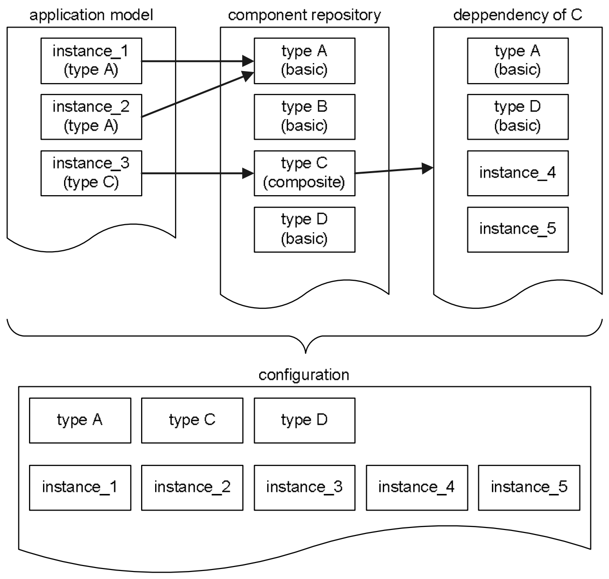

Figure 8. Configuration through selection

When the system is updated, in most cases, only a small portion of an application is changed, typically involving few components. Accordingly, when the application is reconfigured, the binary executable is modified as well. Therefore, the goal of reconfiguration is to find out the differences between the new configuration and the previous configuration. The difference can be generated to denote which components are added, removed, updated through comparison of two configurations. Based on the difference information, the corresponding components can be added, removed, updated from the executable binary. Changes can be patched to the previous binary in order to avoid assembling all components again.

The process of configuration and reconfiguration can be actually combined into one process. If it is the first-time configuration, an empty configuration is used to be compared to the current configuration, as there is no previous configuration, see Figure 9. From the point of view of implementation, whatever COMDES function blocks are designed, they are fundamentally no more than relocatable objects, which are compiled from source files for specific architectures and can be linked to an executable in terms of linker script [6].

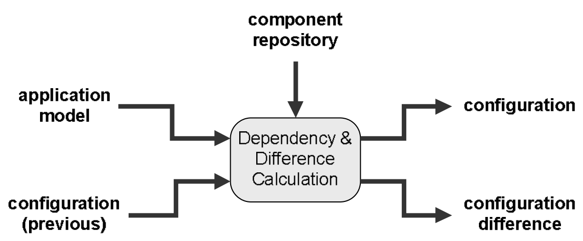

Figure 9. Dependency and configuration difference calculation process

The configuration and reconfiguration are just concerned about how to create an executable from these objects by putting them into specified locations. This stage is called component assembly.

4 Component assembly

Figure 10 presents the standard development process of programs written in C. The source files and include files are compiled by the compiler/assembler into relocatable object files. The linker combines the object files with standard libraries into the executable code and maps them to specific memory locations defined in the linker script.

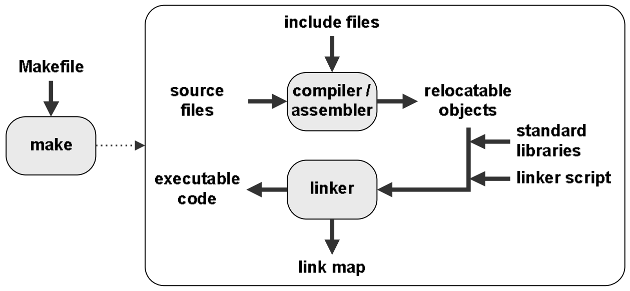

Figure 10. Conventional software development process

The link map contains information about the memory allocation of objects in the executable code. The whole process is directed by the *make* utility operating according to instructions given in Makefile.

The standard libraries are standardized collections of include files and routines used to implement common operations, such as mathematical functions and string manipulation.

The final executable code of a COMDES application is said to be configured, because it consists of predefined reusable components in the form of data structures and executable functions. The former represent component instances and the latter – component types; each particular instance has a corresponding data structure and is associated by its type with a number of functions. In this way, during application synthesis no executable code is created but only data structures are instantiated, which 'glue' components together. Figure 11 illustrates the development process of COMDES configuration.

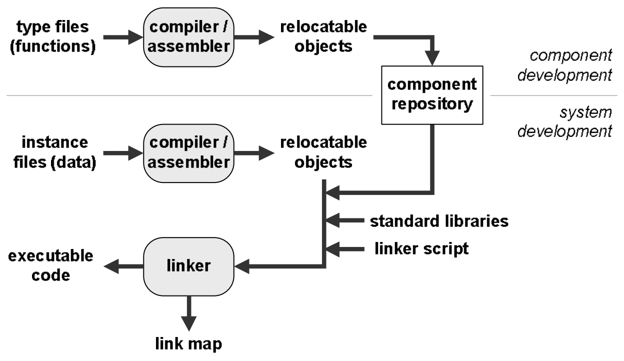

Figure 11. Overview of system configuration out of prefabricated components

This approach is possible thanks to the design of COMDES component design patterns, which consist of an interface and an implementation providing component independence and thus, a way to update a function block without influencing others function blocks. Patterns are implemented in plain C language following the principle of modular design with limited object orientation [7, 8].

The glue codes concerned with the instances of a configuration are generated by a code generator, which is not a topic of this paper. Besides glue codes, there are two files used in order to build the application binary: Makefile, which specifies what components should be linked, and linker script, which specifies where components should be placed in memory. The generation of these files is one of the tasks of component assembly.

Another significant characteristic of a COMDES system is its static configuration. This means that the system is entirely defined during the design stage: there is no dynamic memory allocation, the final code is statically linked, which not only helps to avoid any run-time overhead, but also eliminates possible run-time errors like out-of-memory and unresolved references.

The reconfiguration process is the same as the configuration process except the linking part, where the actual building of an application takes place. Therefore, the component instances of the updated application are compiled into object files, as usual. Next, the reconfiguration linking process takes place, as depicted in Figure 12:

- Pre-linking: It is the regular linking of all objects in order to find (library) dependencies and to produce a link map. This step is needed to detect changes in the application.
- Compare link maps: The pre-linking link map is compared with the previous version of the application link map. As a result, modified/new and removed library objects are detected. Remark: information of modified/new and removed function block instances and types comes from the difference file generated in the component selection process.
- Generate symbol file: A file containing all previous symbols that are present in the update is produced.
- Strip symbols: Removed objects are stripped from the previous executable code.
- Generate linker script: The reconfiguration process heavily relies on the linker script functionality that controls object placement in the memory. This step produces a new linker script according to a defined application memory layout. The goal is to minimize changes in the updated code.
- Linking: The final step produces the reconfigured executable code, which could be the whole code or just a patch.

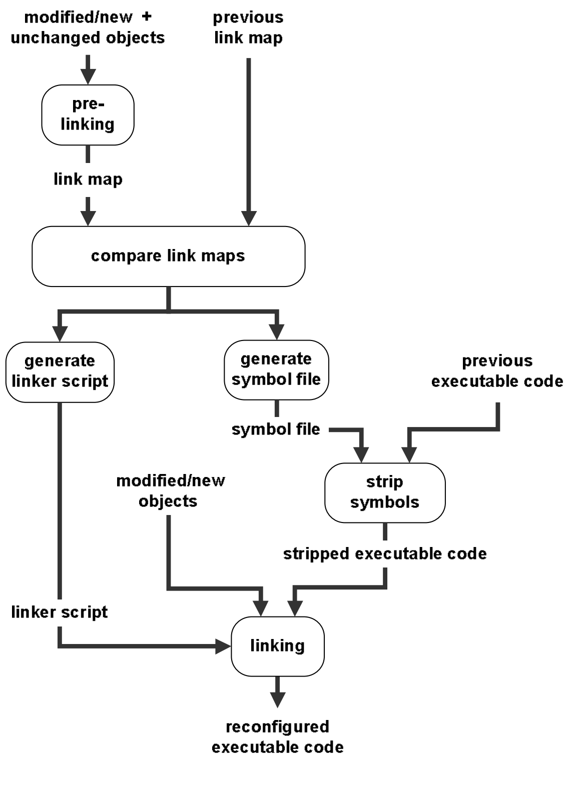

Figure 12. Overview of the reconfiguration linking process

The presented linking process is an implementation of the concept of incremental linking with exact positioning of component code.

5 Conclusion

COMDES is a domain-specific framework for real-time distributed control systems. It employs modeling techniques that reflect the nature of embedded systems, which are predominantly real-time control and monitoring systems, i.e. function block diagrams. The COMDES design method is essentially concerned with the specification and configuration of embedded systems in a computer-aided environment. This paper has presented two steps of the configuration process: component selection and component assembly, which will ultimately allow for automatic configuration of applications from prefabricated components.

The COMDES architecture supports reconfiguration, which is an important issue in the area of embedded systems. The latter are predominantly dedicated systems but they must be able to adapt to changes in requirements, and to facilitate the removal of software errors. The adopted reconfiguration technique allows for the generation of the entire executable code (with minimal changes) or only the modified part of it. The latter feature makes code patching possible, which minimizes the time needed for applying the update.

6 References

[1] C. Angelov, Xu Ke and K. Sierszecki, "A Component-Based Framework for Distributed Control Systems", Proc. of the 32nd EUROMICRO Conference SEAA 2006

[2] Xu Ke, K. Sierszecki, "Generative Programming for a Component-Based Framework of Distributed Embedded Systems", Proc. of the 6th OOPSLA Workshop on Domain-Specific Modeling, 2006

[3] A. Zoitl, at al., "Towards Basic Real-Time Reconfiguration Services for Next Generation Zero-Downtime Automation Systems", in International IMS Forum: Global Challenges in Manufacturing, 2004

[4] H. J. Reekie and E. A. Lee, "Lightweight Component Models for Embedded Systems", Tech. Report UCB ERL M02/30, Electronics Research Laboratory, University of California at Berkeley, 2002

[5] Magnus Larsson, "Applying Configuration Management Techniques to Component-Based Systems", ISSN 1404-5117, MRTC Report 00/24

[6] J. R. Levine, "Linkers and Loaders", 1999

[7] J. R. Hayes, "Modular Programming in C", Embedded Systems Programming, 14, 2001

[8] M. Richardson, "Object-Based Development Using the UML and C", Dedicated Systems Magazine, Q1 2001