A Framework for Generation of Inter-node Communication in Component-based Distributed Embedded Systems

Luka Lednicki, Jan Carlson
Mälardalen Real-time Research Centre
Mälardalen University
Västerås, Sweden
luka.lednicki@mdh.se, jan.carlson@mdh.se

Abstract

In component-based and model-driven development it is common to model embedded applications in a platform-independent manner. As an example, some approaches allow development of distributed applications while abstracting away from details of communication between platform nodes. Using such an approach requires to implement this communication before an executable system is deployed. Currently it is common to automatically implement this communication on the level of code, while providing it on the model level is mostly a task that needs to be done manually. In this paper we present a framework for automatic generation of inter-node communication by adding communication components to software models. The framework provides flexibility in the level of automation of generation decisions, and is defined in a way which allows adding support for new communication media or protocols. We have implemented the generation framework for the IEC 61499 standard and provide a prototype generation tool, which we use for examining the applicability of the approach.

1 Introduction

Component-based [1, 5] and model-driven [4] development approaches have shown potential in improving the process of development of distributed embedded systems. A common feature that these approaches tend to provide is development of systems using platform-independent software models, reducing the amount of details that a developer needs to tend to during early stages of development. Later in the development, these platform-independent models can be transformed to either platform-specific models or synthesized directly to platform-specific code. Although it is common to generate platform-specific code without using intermediate models, generating a platform-specific system model can provide additional benefits to development. For example, platform-specific models can make developers more aware of consequences of deployment decisions, or make platform-specific properties available to model-level analysis tools.

An example of an approach which supports separation of platform-independent and platform-specific models can be seen in the IEC 61499 standard [11, 16, 17]. The standard allows platform-independent development of distributed applications, abstracting away some of the additional complexity of communication between distributed nodes. While some development tools [9, 14] automatically generate code for this communication when deploying an application to hardware, inter-node communication can also be implemented using specialized communication components on the model level. These components currently need to be added to the models and updated manually, thus extending the development time and increasing the possibility of introducing errors.

In this paper we present a framework for automatic generation of inter-node communication on the level of software models, and provide an implementation of the framework for the IEC 61499 standard. The generation is performed by utilizing the models of software, platform and the mapping between the two. The method first derives a communication model of the system and determines available communication media. After that, communication media and protocols to be used for implementing the communication are selected. In the end, the communication components are added to the software model and configured for communication, and the original software model is annotated with information about the generated elements.. We also provide an prototype generation tool and use it to demonstrate the applicability of the approach.

The approach allows developers to model distributed functionality without having to manually implement the details of inter-node communication. The communication model elements can be generated even in the early stages of development, before the system is fully implemented, which makes it easier to explore different allocation options. To increase flexibility and make it easier to apply the method do different component frameworks, we

separate the generation into multiple phases, each with clearly defined inputs and outputs. The framework also distinguishes between the generic generation mechanism and generation of protocol-specific elements, allowing the method to be extended with support for different protocols.

The rest of the paper is organized as follows: In Section 2 we provide background information on the IEC 61499 standard. Section 3 contains detailed description of the communication generation framework. In Section 4 we show how the generation method is applied on an example system, while in Section 5 we provide information about the prototype implementation of our method. Section 6 gives an overview of related work and Section 7 concludes the paper.

2 Background - IEC 61499

The IEC 61499 standard [11, 16, 17] is proposed as a successor of the IEC 61131-3 standard [10] widely used in industry to accommodate development of industrial automation systems. One of the main concerns of the standard is to allow development of applications distributed over multiple controllers, which can be geographically separated in a plant. In the following two sections we will describe the software, platform and resource-specific software model of the standard.

2.1 Software model

The main element of the IEC 61499 software model is the *function block*. Function blocks are reusable units of software that implement a specific functionality with a clear separation between interface and implementation.

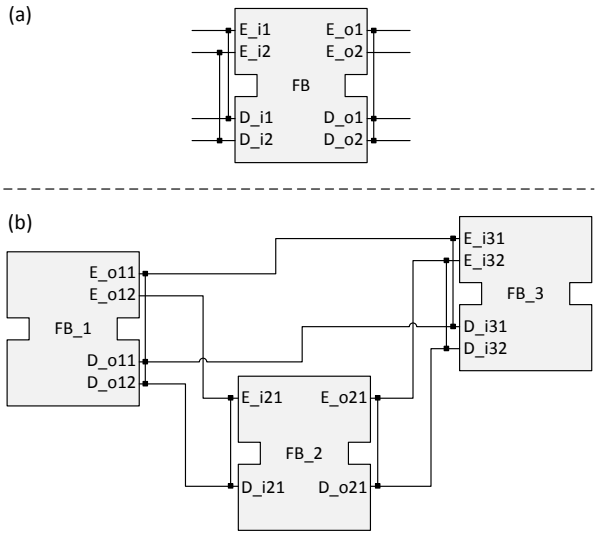


Figure 1. a) An IEC 61499 function block interface. b) An application built using a function block network.

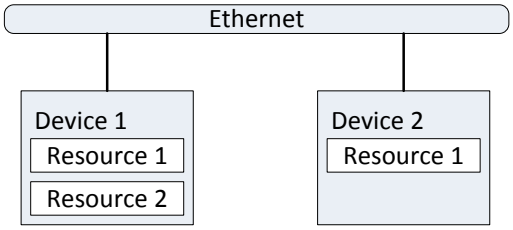


Figure 2. Example model of the platform in IEC 61499.

Although the functionality of function blocks can be implemented in various ways, the work presented in this paper deals with function blocks only on the interface level. Therefore in this section we will not describe the details of function block implementation.

The *function block interface* defines how the functionality of a function block is presented to the rest of the system. The interface explicitly separates event and data inputs and outputs. Event inputs and outputs are used to specify the execution flow, but do not provide any means for exchanging data between function blocks. All data transfers are done by data inputs and outputs.

Relations between event and data ports can be described by *WITH qualifiers*. Defining a WITH qualifier on an event input port and a set of data input ports describes which data inputs will be sampled together with the event port. A WITH qualifier combining an event output port with a set of data output ports shows which data outputs will be updated with new values together with an output at the event output.

Figure 1 (a) shows an example of a function block interface. The figure shows a function block with input event ports *E_i1* and *E_i2*, output event ports *E_o1* and *E_o2*, data inputs *D_i1* and *D_i2*, and data outputs *D_o1* and *D_o2*. The WITH qualifiers are represented by connecting the ports with vertical lines, marking each port belonging to a WITH qualifier by a black rectangle. As an example, the WITH operator defined on outputs of *FB* implies that when an event is generated on *E_o1*, the values on ports *D_o1* and *D_o2* will also be updated.

IEC 61499 *applications* are implemented by *function block networks*. A network consists of a set of function blocks and connections between the ports of these function blocks. Applications provide a view of the complete software implementation, no matter how the function block implementing them are mapped to the platform nodes.

An example of an application containing function blocks *FB_1*, *FB_2* and *FB_3*, and connections between them, is given in Figure 1 (b).

2.2 Platform model

In IEC 61499 the platform is represented by *devices*. A device is an independent physical entity capable of per-

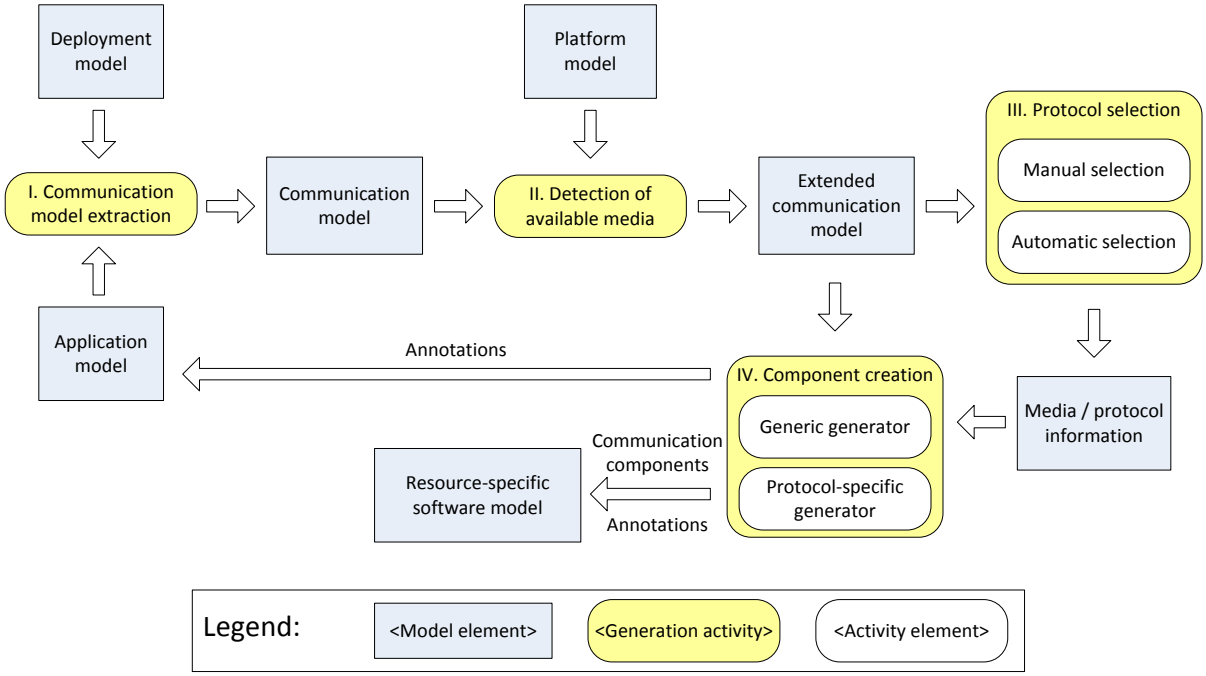


Figure 3. Overview of the communication component generation process.

forming one or more specified functions. Each device contains one or more *resources*, which are functional units with independent control of operation.

Communication networks between devices are modeled by *network segments*. One device can be connected to more than one segment, and there is no limit to the number of devices that can be connected to a single network segment.

Figure 2 shows an example of platform model consisting of two devices connected via an Ethernet network segment. *Device 1* has two resources, and *Device 2* with only one resource.

Applications are deployed to the platform by mapping its function blocks to the resources contained by the devices of the platform. Since function blocks are atomic units of deployment, each function block can be mapped to only one resource.

2.3 Resource-specific software model

Each resource in a distributed IEC 61499 system contains a local model of the application, containing only a subset of applications function blocks which are mapped to that particular resource. This model can however contain additional function blocks which are not visible on the application level. It is a common practice to use this option to implement functionality that is specific to the current mapping of the application, such as adding inter-resource communication function blocks.

3 Communication generation framework

To solve the problem of automatic generation of inter-node communication components we define a generation framework, an overview of which is depicted in Figure 3.

The framework is separated into four activities, each with clearly defined inputs and outputs, introducing a level of flexibility which results in multiple benefits. First, the method can easily be extended to take into account more information, for example to improve media and protocol selection, or the component creation method. Arbitrary new communication components, media and protocols can easily be appended to the generation implementation. The level of automation of the generation process can be varied, leaving the ability for manual input of a developer. Also, the separation allows easier adaptation of the method to different component frameworks.

In the following sections we first introduce the communication model which is used to describe inter-node communication in our approach, and then give detailed descriptions of the four generation phases.

3.1 The communication model

We describe the communication between components located on different nodes by creating a communication model. The Ecore metamodel of the communication model can be seen in Figure 4. The main elements of the model are *Channels*, which represent data or events produced together on the same source node, and transferred as messages to one or more destination nodes.

The content of a message sent through a channel is defined by a set of *Data* elements. Each data element defines the type of data it represents using the *type* attribute.As events are not distinguished by types, and carry no semantics besides the occurrence of an event, we represent them by *Data elements* with *type* set to *none*.

In addition to a set of data elements, a channel also con-

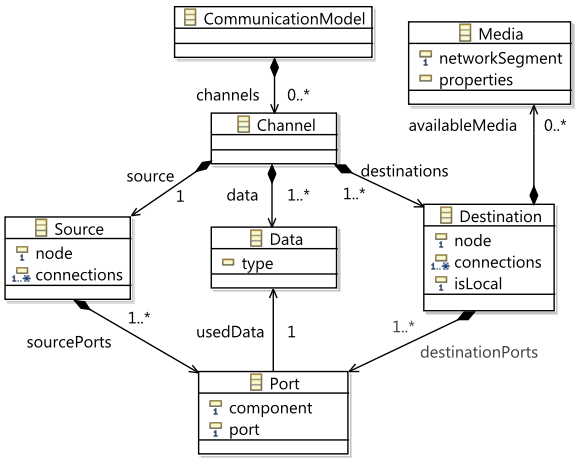


Figure 4. Ecore metamodel that defines the communication model.

tains one *Source* and one or more *Destination* elements. Both of these element types have a reference to the platform node which is the endpoint of the communication. The *Destination* element has an additional *isLocal* flag, which indicates if the source and destination nodes are two virtual nodes residing on the same physical node, or if they are distributed on separate physical nodes. As an example, in the IEC 61499 implementation of the generation method, communication between two resources belonging to a same device is marked as local. The *Source* and *Destination* elements also reference one or more inter-node connections in the application model which they represent using the *connections* attribute.

The source and destination elements also describe how the messages are created and consumed by components. This is done by a set of *Port* elements which are added either to the *sourcePorts* set of a source or the *destinationPorts* set of a destination. Each of these elements define from which port of which component in the application model the message data is read (in case it is added to a source) or to which port of which component the data needs to be delivered (in case it is added to a destination). This is done using the *component* and *port* attributes. The message element that is generated by or delivered to the referenced port is denoted by a reference to a *Data* element of the *Channel*.

Each *Destination* element can also contain a number of *Media* elements. These elements are used to describe which communication media can be used for communication between the destination and the source of the channel. Besides the *networkSegment* attribute which references the network segment of the platform model, a *Media* element also contains a *properties* attribute which can contain information about how the destination node is connected to the network segment, for example an IP address of a device on an Ethernet network.

3.2 Communication model extraction

The communication model of an application is extracted from the application model and the deployment model. The process starts with detecting event and data connections in the application which connect components deployed to different platform nodes. A *Channel* element is generated for each group of connections for which source events and data are generated at the same time and on the same node. Then, the *Data* element set of the channel is generated based on the data end event sources of the connections. The channels single *Source* element is created and initialized with the information about the source node and the represented connection, and *Port* elements of the *sourcePorts* set are created based on the output ports from which the connections represented by the channel start.

The represented connections are then grouped by the nodes that their destination components are mapped to. For each such group, a *Destination* element is added to the channel. A *Port* element for each connection is then added to the *destinationPorts* set, and initialized to point to the target port of the connection.

In the IEC 61499 implementation of the generation method, we use WITH operators defined on the output ports to group connections into channels, as they define one event and an arbitrary number of data outputs generated at the same time.

3.3 Communication media detection

Once the communication model has been derived, it can be used to determine which media alternatives are available to implement the inter-node communication. This is done in combination with the model of the platform.

As channels can have multiple destinations, each on a distinct platform node, media detection has to be performed separately for each channel destination. Available media for a destination is determined by finding all communication networks in the platform model to which both the node of the destination and the node of the channel source are connected. For each available media, a *Media* element is added to the *Destination* and the value of the *properties* attribute for the new element is set based on how the node is connected to the network.

The results of the media detection are added to the existing communication model and the new extended model is stored.

3.4 Protocol selection

After the media detection is done, it must be decided which of the available media will be used to implement the communication, and how (i.e. using which protocol). The protocols which can be used to achieve communication using a specific media are defined by the available protocol specific generators, which will be described in the next section. This process can be a combination of manual and automated selection. While manual selection allows de-

velopers to use expert knowledge to obtain a desired system behavior, automated selection allows optimizations on the system level and faster development in cases when there are no specific communication constraints to consider.

As an example of optimization during automated selection, we have implemented an algorithm which reduces the number of used communication media. The algorithm finds common media for destination of a communication channel, and selects the ones which cover the most destinations. In this way the number of generated components and sent messages will be minimized.

3.5 Communication component creation

The actual creation of the communication components is based on the extended communication model and the information about selected communication protocols. It is done using two mechanisms: a *generic generator* and a set of *protocol specific generators*.

The task of the *generic generator* is to traverse the communication model and initiate creation of communication components for sources and destinations. For each destination exactly one component is generated on the destination node. Creation of components on the source side is more complex. Messages from one source can be delivered to more than one destination, and each destination can require different communication media or protocol. Because of this, in some cases there is a need to generate more than one communication component for a single communication channel source, each for a specific media or protocol. After the communication components are generated, the *generic generator* creates connections between applications components and the generated communication components based on the information stored in the *Port* elements of the communication model. In the end, the information about the connections represented by sources and destinations is used to annotate these connections with the information about the generated components.

The component creation initiated by the *generic generator*, as well as configuration of the created components, is performed by a *protocol specific generator*. The generation implementation can have more than one specific generator, each implementing component creation for a specific communication media and protocol. The specific generator to be used for each source and destination is chosen based on information provided by the protocol selection phase.

In the IEC 61499 context, we have implemented the *generic generator* and a *protocol specific generator* for communication using the UDP protocol on an Ethernet network. This type of communication is part of an interoperability provisions defined by Holobloc [8]. In the resource-specific software model, the communication is implemented using two standardized function block types: the PUBLISH function blocks are used to send multicast UDP messages to the network, while SUBSCRIBE function blocks receive such multicast messages. There exist multiple versions of both function block types, each with a different number of data values they send or receive. To enable sending messages from one PUBLISH function block to one or more SUBSCRIBE function blocks, they must be configured to send and receive messages on the same UDP port.

The *specific generator* for UDP over Ethernet generates PUBLISH function blocks for channel sources, and SUBSCRIBE blocks for destinations, choosing the correct versions of the function blocks based on the number of data values transferred by the channel[1]. Each channel is then assigned a unique UDP port which is used to configure all communication blocks belonging to the channel which will communicate using this protocol.

For the purpose of testing the generation of communication for different media and protocol types, we have also created a protocol specific generator for communication using a CAN bus. However, implementation of components for such communication does not exist in the IEC 61499 development tools which we used to build the prototype generation tool. Because of this, the CAN protocol specific generator can only be used for testing purposes.

4 Example

We will illustrate the proposed communication component generation method on an example system. The distributed application of the example is shown in Figure 5 a), and the platform model is given in Figure 5 b). Function block *FB_1* is mapped to run on *Resource 1*, *FB_2* and *FB_3* to *Resource 2*, and *FB_4* to *Resource 3*.

Figure 5 c) shows the communication model extracted from the application and deployment models. The extraction results in two communication channels, one for each WITH qualifier of *FB_1*. The first channel is used to transmit one event and two data values from *Resource 1* to a single destination on *Resource 2*. The model also shows at which port the message data originates, and which destination ports receive the data. The second channel transmits one event and one data value originating on *Resource 1*, and has two destinations, one on *Resource 2* and one on *Resource 3*.

Results from communication media detection are also depicted in Figure 5 c). Both *Destination 1* of *Channel 1* and *Destination 1* of *Channel 2* can be reached from their sources by either the CAN bus or the Ethernet connection. Ethernet is however the only media that can be used to communicate between the source of *Channel 2* and *Destination 2* of the same channel.

We assume that the developer has manually selected the CAN bus as the media to be used for *Destination 1* of

[1] Because of the specifics of the implementation of message decoding in SUBSCRIBE function blocks, in some cases where not all outputs are used in all destinations, the method generates multiple PUBLISH blocks and treats each of these as a separate communication channel.

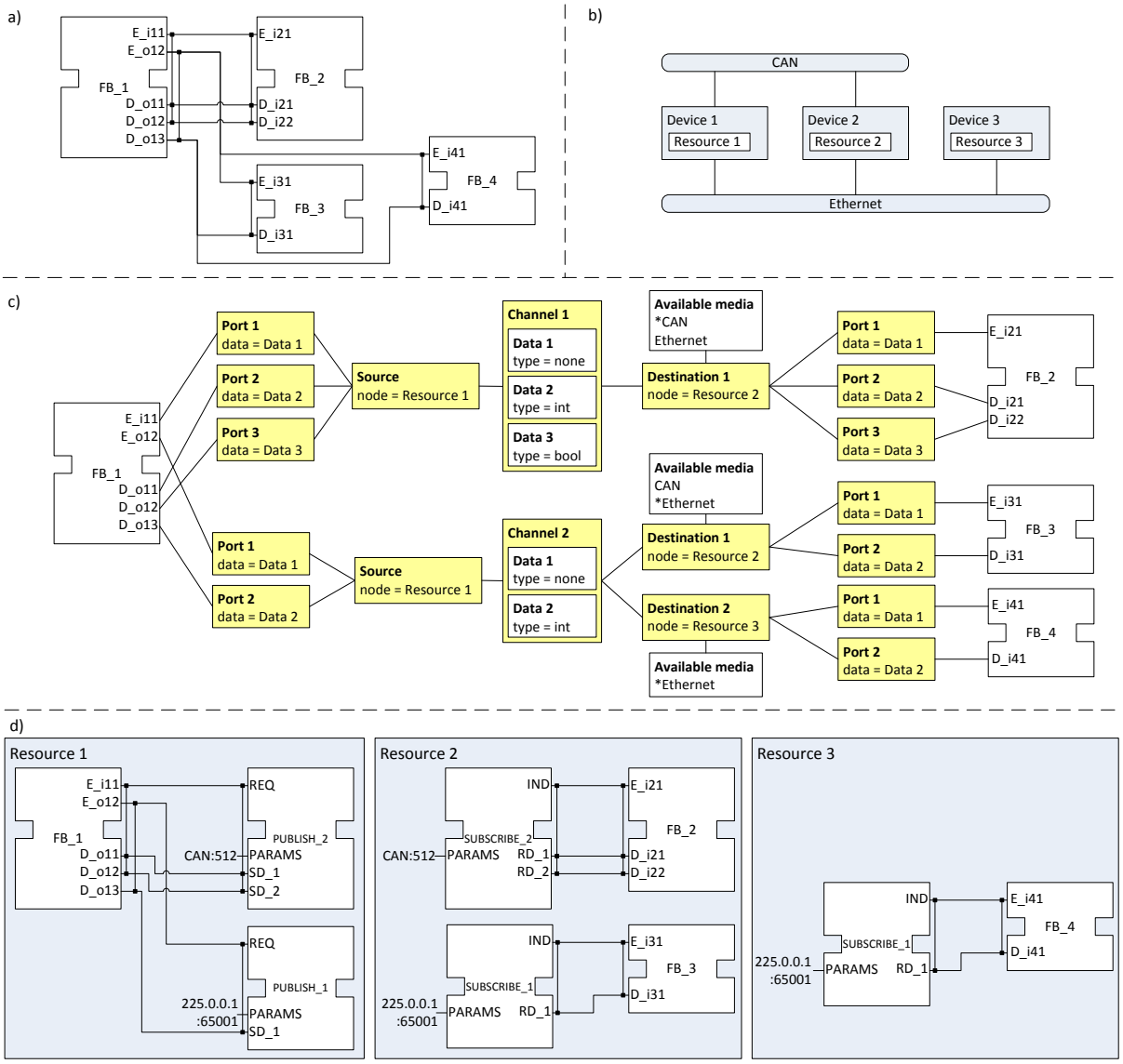


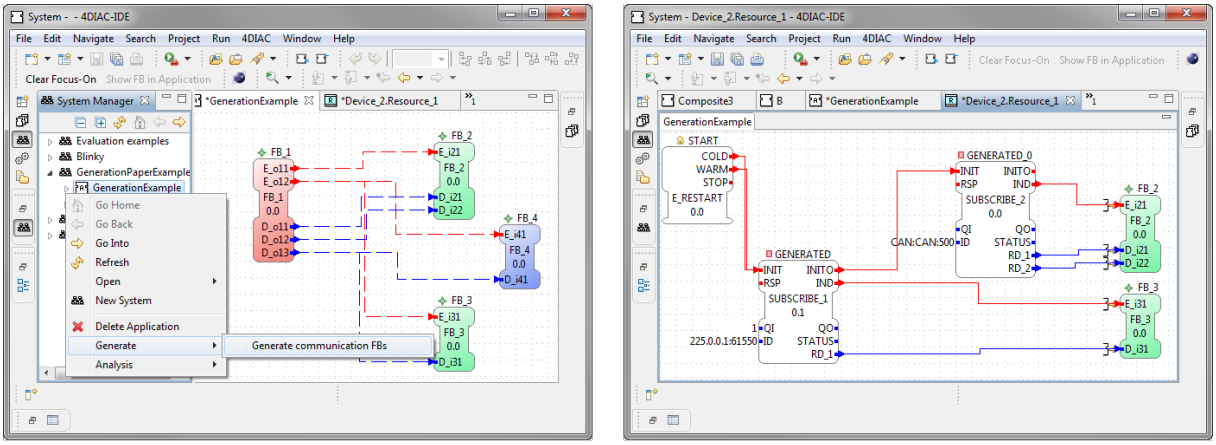
Figure 5. a) Example application model. b) Platform model of the example system. c) Communication model derived from the example application and platform model. d) Resource-specific software models containing generated communication components.

Channel 1, and that media selection for *Channel 2* is left to the tool. As Ethernet is a common available media to all destinations of *Channel 2*, it is selected to implement communication for both destinations by the automated selection process. The selected media are marked with an asterisk in the model figure.

The result of the component creation can be seen in Figure 5 d). The figure shows resource-specific software models of the system resources, which, in addition to the deployed function blocks, contain the inter-node communication function blocks. For the sake of simplicity, the models do not show all ports of PUBLISH and SUBSCRIBE function blocks, such as the ports used to trigger function block initialization.

5 Implementation

To demonstrate the applicability of the communication component generation method we have implemented it as a prototype tool. The tool has been developed as a plug-in for the 4DIAC-IDE, an open-source IEC 61499 development environment [15]. Integration with 4DIAC allows us to execute the tool using the graphical model editors, and perform generation using existing system models. Generation results in systems which are fully executable without any need for manual editing of resource-specific software models or code. A screenshot depicting the 4DIAC-IDE is shown in Figure 6. The figure contains a) the application model from the previous example and the generation menu added by our plug-in, and b) the resource-specific



a) b)

Figure 6. Screenshot of a) example application and the automatic generation menu in 4DIAC-IDE and b) a resource-specific software model containing components created by the prototype generation tool.

software model for *Resource 2* containing the generated communication blocks. The prototype tool is freely available for download[2].

6 Related work

The general problem of code generation using system models has been explored by the model-driven development research community, and some of the approaches also allow automatic generation of distributed systems.

In Balasubramanian et al. [2] describe how models of software, platform and the mapping between the two can be used to generate parts of code that implement access to the communication media. Gokhale et al. [7] describe how embedded systems where communication between nodes is supported by middleware can be generated using system models. The authors propose using platform-independent application models together with the models of platform to configure pre-existing middleware components. Compared to our generation, which generates communication on the level of models, the communication that both of these approaches generate is not visible to the developers on the model-level, and not available to analysis methods which are performed using models.

Automatic generation of inter-node communication on the code level has been implemented in ISaGRAF [9] and nxtStudio [14] development tools for IEC 61499 systems. Our approach however generates this communication on the model level, making it more visible to the system developers and analysis tools. In our approach we have also introduced separate generation phases, all of which have well-defined inputs and outputs, making the generation easily adaptable to new communication protocols, or transferable to other component-based frameworks.

Doukas and Thramboulidis [6] present a real-time framework that is able to run systems created using function block models, which also includes automatic generation of inter-device communication. The communication is implemented using entities called event-connection managers, which allow the communication to be more flexible than implementation using only code. Compared to our approach, the automatically generated communication is not visible in function block models, the generation takes into account only the deployment model as opposed to the complete platform model, and communication can only be implemented using IPCP protocol.

Brisolara et al. [3] provide generation of communication on the level of models as a part of a method which uses high-level UML models to generates executable and synthesizable Simulink models. Providing extensive support for automatic generation of communication was not the main aim of this work. Compared to work presented in this paper, the communication generation does not take into account the model of platform nodes and network connections between them, and therefore can not generate communication for different communication media and protocols. Also, in this approach the information about generated elements is not propagated back to the UML model.

7 Conclusion

In this paper we have defined a framework for automatic generation of inter-node communication in component-based distributed embedded applications. The generation is done by first extracting an inter-node communication model from the models of software, platform and deployment. Based on this model, the software model is updated by adding, configuring and interconnecting components which will implement communica-

[2] http://www.idt.mdh.se/~jcn01/research/4DIAC-plugins/

tion between distributed platform nodes. The generation method is separated in multiple stages, each with clearly defined inputs and outputs. The result of this separation is a flexible and extensible framework which can easily be updated with support for new communication medias and protocols, or applied to different component models. The framework also allows a variable level of automation of the communication media and protocol selection in case the communication can be implemented using multiple protocols. We have described how the generation framework can be applied to the IEC 61499 standard and demonstrated the generation on an example system. The framework has also been implemented in form of a prototype tool, based on the 4DIAC integrated development environment for IEC 61499 systems.

As future work we would like to validate the generation framework in a realistic case-study.

We also plan to extend the presented generation method with a possibility of creating relay communication components. This approach would enable communication between two nodes which are not directly connected by a network, but share a common node to which both are connected.

Another possibility for future work is to investigate how the generated communication components and annotations to the application model could be used to improve the results obtained by model level analysis, such as described in [13, 12].

Acknowledgments

This work has been performed as part of the Ralf3 project founded by the Swedish Fundation for Strategic Research.

References

[1] C. Atkinson, C. Bunse, C. Peper, and H.-G. Gross. Component-based software development for embedded systems – an introduction. In *Component-Based Software Development for Embedded Systems*, pages 1–7. Springer, 2005.

[2] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, Feb 2006.

[3] L. B. Brisolara, M. F. S. Oliveira, R. Redin, L. C. Lamb, L. Carro, and F. Wagner. Using UML as Front-end for Heterogeneous Software Code Generation Strategies. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 504–509, New York, NY, USA, 2008. ACM.

[4] S. Burmester, H. Giese, and W. Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Model Driven Architecture–Foundations and Applications*, pages 25–40. Springer, 2005.

[5] I. Crnkovic and M. Larsson. *Building reliable component-based software systems*. Artech House Publishers, 2002.

[6] G. Doukas and K. Thramboulidis. A Real-Time-Linux-Based Framework for Model-Driven Engineering in Control and Automation. *Industrial Electronics, IEEE Transactions on*, 58(3):914–924, March 2011.

[7] A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming*, 73(1):39 – 58, 2008. Special Issue on Foundations and Applications of Model Driven Architecture (MDA).

[8] Holobloc Inc. Function block development kit (FBDK), May 2012. http://www.holobloc.org/.

[9] ICS Triplex ISaGRAF. ISaGRAF, 2014. http://www.isagraf.com/.

[10] IEC 61131-3: Programmable Controllers–Part 3: Programming Languages. International Electrotechnical Commission, Geneva, 1993.

[11] IEC 61499-1: Function Blocks-Part 1 Architecture. International Electrotechnical Commission, Geneva, 2005.

[12] L. Lednicki, J. Carlson, and K. Sandström. Device utilization analysis for IEC 61499 systems in early stages of development. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–8, 2013.

[13] L. Lednicki, J. Carlson, and K. Sandström. Model Level Worst-case Execution Time Analysis for IEC 61499. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 169–178, New York, NY, USA, 2013. ACM.

[14] nxtControl. nxtStudio, 2014. http://www.nxtcontrol.com/.

[15] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sünder, A. Valentini, and A. Martel. Framework for Distributed Industrial Automation and Control (4DIAC). In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 283 –288, july 2008.

[16] V. Vyatkin. IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review. *Industrial Informatics, IEEE Transactions on*, 7(4):768 –781, nov. 2011.

[17] A. Zoitl, T. Strasser, K. Hall, R. Staron, C. Sünder, and B. Favre-Bulle. The past, present, and future of IEC 61499. *Holonic and Multi-Agent Systems for Manufacturing*, pages 1–14, 2007.