Model-to-Metamodel Transformation for the Development of Component-Based Systems

Gerd Kainz[1], Christian Buckl[1], Stephan Sommer[2], and Alois Knoll[2]

[1] fortiss, Cyber-Physical Systems,
Guerickestr. 25, 80805 Munich, Germany
{kainz,buckl}@fortiss.org
[2] Faculty of Informatics, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
{sommerst,knoll}@in.tum.de

Abstract. Embedded systems are a potential application area for component-based development approaches. They can be assembled from multiple generic components that can either be application components used to realize the application logic or hardware components to provide low level hardware access. The glue code to connect these components is typically implemented using middleware or run-time systems. Nowadays great parts of the system are automatically generated and configured according to application needs by using model driven software development tools. In a model driven development process, three different kinds of developers can be identified: run-time system experts, component developers and application developers. This paper presents a multi-phase approach, which is suited to support all of these experts in an optimal way. Key idea is a multi-phase development process based on model-to-metamodel transformations connecting the different phases. The advantages of this approach are demonstrated in the context of distributed sensor / actuator systems.

Keywords: Model-to-Metamodel (M2MM), Model-to-Model (M2M), Model Transformation, Component-based Systems, Model-Driven Development (MDD).

1 Introduction

During the development of embedded systems, many recurrent tasks have to be implemented. To increase the development efficiency of distributed embedded systems, a component-based architecture can be used to raise the level of reuse [1]. In such systems, the underlying software architecture is based on a component / container architecture: the container in form of a middleware architecture or run-time system implements the communication between the different components. Although generic approaches, e.g. CORBA [2], are available and used in the area of distributed standard IT systems, domain specific run-time systems / middleware are dominant in the area of distributed embedded systems [3] to adapt to special requirements in this domain, e.g. resource limitations. To

D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.): MODELS 2010, Part II, LNCS 6395, pp. 391–405, 2010.

speed up the adaptation process of these run-time systems, model-driven methods can be used to automatically generate a tailored run-time system for a specific application. Code generation can also take non-functional requirements into account. One example is to hide the heterogeneity of different communication protocols by using a specific middleware. To guarantee the required quality of service (QoS) even if the used communication protocol does not provide means to guarantee them, the middleware might require additional functionality on top of the used communication protocol.

When analyzing the development process of embedded systems, three different kinds of developers can be identified: run-time system experts, component developers, and application developers. The experts for the run-time system develop generic architectures, which are later automatically tailored by the development tool according to the requirements of a concrete application. Component developers are responsible for the creation of both software and hardware components. They have to ensure that the developed components can be integrated into the run-time system, either by providing appropriate interfaces for the software components or by offering a suitable firmware for hardware components. Finally, application developers select the right components for their application and specify all non-functional requirements, which have to be satisfied by the running system, using model-based tools.

Currently, model driven development has been mainly applied to support the application developers in creating new applications focusing on the application logic. In this paper, we present an approach, which consecutively assists all three developer types. This is achieved by a three-phase model-driven development approach, where every phase builds up on the input of the previous phase. In each phase the corresponding expert is able to model the relevant properties of the system and to concentrate on the aspects of the system related to his expertise. The different phases are connected via model-to-metamodel (M2MM) transformations, where objects of the previous phase become classes in the subsequent phase. This paper gives an overview of the applied approach and describes the results of a first prototype.

The remainder of this paper is as follows. Section 2 shortly describes the domain used for evaluation and gives a motivation for the approach. The proposed operational sequence is outlined in section 3 and compared with corresponding related work in section 4. At the end, section 5 summarizes the contribution and gives an outlook on further steps.

2 Application Domain: Distributed Sensor / Actuator Systems

Distributed sensor / actuator networks are an interesting application area for the component-based approach, as many different generic components, e.g. for sensing, actuation or communication, can be identified. The ϵSOA project [4] targets this application area. In the scope of the project, a model driven development tool for distributed sensor / actuator systems has been developed to simplify the development of such system.

One main property of sensor / actuator networks is the high level of hardware heterogeneity. In the current version Linux-based computers, wireless sensor nodes with TinyOS[1] and Atmel 8 bit micro controllers without an operating system are supported. In addition, different types of sensors and actuators are available for the interaction with the environment. Figure 1 shows a demonstrator implemented within the ϵSOA project.

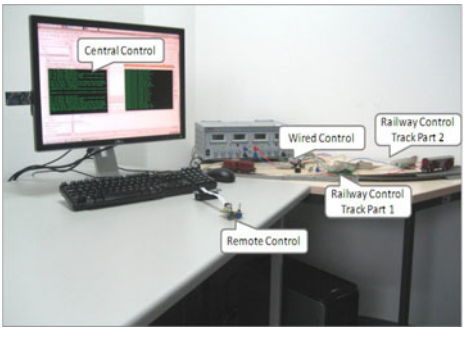

Fig. 1. Model Railway Controlled by Services Running on Distributed Heterogeneous Controllers

By using a service oriented architecture (SOA) the heterogeneity can be hidden from the application developer [5]. The system is interpreted as a set of services. These services can either be hardware-related components implementing the interaction with sensors respectively actuators or components used to realize hardware-independent calculations. With the development tool created in the project, the end user is able to select all the services required for the application, assign them to concrete hardware nodes and configure them accordingly. The middleware needed for the execution is generated by a model-driven code generator. The code generator takes care that only the required functionality is integrated into the middleware and that the middleware is configured and tailored to fit the actual application requirements.

Due to the high diversity of sensors, actuators and controllers, the development tool had to be extended several times. Examples are the addition of new services used to realize control algorithms or logics. Furthermore, the support of new hardware platforms caused many changes. The effort for the integration turned out to be very high and required in-depth knowledge of the whole development tool. In a concrete case, the system existing of wireless components was extended with components supporting Ethernet communication. Therefore, the metamodel had to be augmented with Ethernet components and their corresponding attributes. During this extension, a specification of the concrete component had been added to the metamodel. Due to this fact, no distinction between

[1] `http://www.tinyos.net/`

the common Ethernet capability with its attributes (e.g. IP address, MAC address, net mask, gateway, ...) and the description of the concrete component had been done. The effect of this approach was that additional effort had to be spent when adding the description of other components with similar / equal capabilities. Moreover this procedure made it difficult to reuse the code generator templates for components with equal capabilities. The observed problem resembles the problem of application development solved by the introduction of the model-driven development tool: large parts of the component descriptions are generic and some of the code generation / configuration logics can be reused in the context of components with similar capabilities. These facts motivate the usage of the model-driven approach and code generation.

Based on this observation, the idea was to introduce a multi-phase model-driven approach: the concept of domain specific development tools to implement systems is adapted to the creation of the development of domain specific development tools themselves. This leads to a speed up in the creation of the development tools. The result is a multi-phase approach, which is elaborated in the next section.

3 Approach

The intention of our approach presented in this paper, is the optimal support of the different developers through a model-driven, multi-phase approach. The basic idea deduces from the general approach of domain specific languages (DSLs) with respect to the model hierarchy, as shown in figure 2. In each phase of the development process, the particular expert can model the relevant fragments. The metamodel of the next phase is then generated through a M2MM transformation.

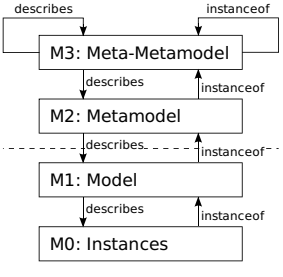

Fig. 2. Model Hierarchy [3]

The benefits of the model-driven approach through DSLs are particularly exploited when used to create similar systems. This is not only restricted to the actual end product shipped to the customer, but also applies for development tools, which shall be constructed. In case of component-based systems, a model-driven approach suits not only for the creation of the end product by combining

the necessary components. It fits in the same way for the development of similar components. The presented approach has been implemented in the context of the ϵSOA project. Currently altogether three phases are supported:

1. Specification of the relevant information for the run-time system by the run-time system experts
2. Description of supported components done by the component producers
3. Configuration of the application by the application developers

These phases are consecutively described further in detail. Afterwards insights into the implementation and code generation steps are given. The approach is used both for hardware and software components. In the context of this paper, we predominantly focus on hardware aspects to have a consistent running example.

3.1 Modeling of the Attributes Used to Describe Components

In the first phase of the approach, run-time system experts specify the information required to describe components used to assemble future applications. Based on the models of the first phase checks, algorithms and steps for code generation are defined. Component unspecific checks can be used to specify tests for correctness of the input models of the third phase. In the example of ϵSOA, e.g. a correct wiring of components only relates to the type of the connected ports (e.g. a temperature input must be connected to a temperature output). By implementing analysis or configuration algorithms, the run-time system experts can define how to calculate a correct schedule or channel assignment / routing. Furthermore, the models of the first phase are used to implement the code generation of the run-time system. This code comprises all the component-unspecific functionality, which normally constitutes the middleware, and serves as a kind of glue code between the different hardware and software components.

Figure 3 shows the underlying metamodel for the description of hardware components. The metamodel allows the run-time system experts to define capability types, e.g. *Communication*, concrete capabilities such as *Ethernet* and associated attributes. By selecting a proper type for each attribute, the possible values can be restricted and wrong configurations can be prevented in advance. Due to this general specification of capabilities, it is possible to implement the

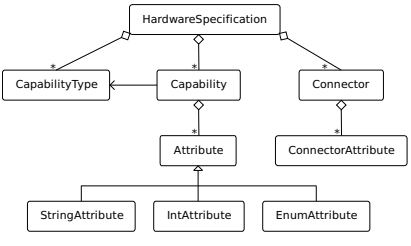

Fig. 3. Metamodel of the First Phase based on Capability Types, Capabilities and their Attributes

run-time system independently from the concrete components used only based on the capabilities and their assigned attribute values. In ϵSOA for example, it can be checked easily, whether the available bandwidth (specified in the attribute *Speed* of *Ethernet*) is sufficient to satisfy the application requirements or not. Since the description of component capabilities is very simple, it can be assured that the underlying metamodel will seldom be changed. This can be compared with the changes made to the metamodel of UML class diagrams. Therefore, the metamodel of the first phase constitutes a solid base for the following steps.

```
<HardwareSpecification>
  <CapabilityType name="ProcessingUnit">
  <CapabilityType name="Communication">
  <Capability name="CPU" type="ProcessingUnit">
    <IntAttribute name="Cores" defaultValue="1"/>
    <StringAttribute name="Type"/>
    <EnumAttribute name="Architecture" defaultValue="x86">
      <value>AVR</value> <value>MSP</value> <value>x86</value> <value>PowerPC</value>
    </EnumAttribute>
    <IntAttribute name="ClockRate"/>
  </Capability>
  <Capability name="Ethernet" type="Communication">
    <StringAttribute name="MacAddress" defaultValue="00:00:00:00:00:00"/>
    <StringAttribute name="IPAddress" defaultValue="192.168.0.1"/>
    <StringAttribute name="NetMask" defaultValue="192.168.0.1"/>
    <StringAttribute name="Gateway" defaultValue="192.168.0.1"/>
    <IntAttribute name="MTU" defaultValue="1024"/>
    <EnumAttribute name="Speed" defaultValue="Mbps100">
      <value>Mbps10</value> <value>Mbps100</value> <value>Gbps1</value>
    </EnumAttribute>
  </Capability>
  <Capability name="RS232" type="Communication">
    <StringAttribute name="Port"/>
    <EnumAttribute name="BaudRate" defaultValue="B38400">
      <value>B9600</value> <value>B19200</value> <value>B38400</value> <value>B115200</value>
    </EnumAttribute>
    <EnumAttribute name="Parity" defaultValue="None">
      <value>None</value> <value>Even_Parity</value> <value>Odd_Parity</value>
    </EnumAttribute>
    <EnumAttribute" name="StopBit" defaultValue="S_1">
      <value>S_1</value> <value>S_1_5</value> <value>S_2</value>
    </EnumAttribute>
    <EnumAttribute name="FlowControl" defaultValue="None">
      <value>None</value> <value>Hardware</value> <value>XON_XOFF</value>
    </EnumAttribute>
  </Capability>
  <Connector name="PCIConnector"/>
</HardwareSpecificaton>
```

Fig. 4. Example Model of the First Phase Defining Processing and Communication Capabilities

An extract of the model used for the ϵSOA project is shown in figure 4. The model defines in addition to the two capability types *ProcessingUnit* and *Communication* the concrete capabilities *CPU*, *Ethernet* and *RS232*. The focus of this model lies in the specification of the basic capabilities with their attributes supported by the run-time system. The great benefit of the three step approach can be seen here. The run-time system experts only need to consider and model the currently supported capabilities and their according attributes. New capabilities or additional attributes can be appended in the future as needed. Thereby the effort for the first modeling iteration can be reduced to an absolute minimum, without complicating any later system extensions. A subsequent enhancement of the metamodel does not lead to inconsistencies in older models of following phases, because adding new capabilities in contrast to deleting existing capabilities does not restrict the validity of old models [6]. In case that additional

attributes need to be added to already existing capabilities, default values can be used to ensure that the code generation for old models is still working and leads to the same result as before. A similar approach has been realized for software components in the ϵSOA project, which are defined by their provided (output) and supported (input) ports.

3.2 Specification of the Available Components

Component suppliers are able to describe their available components and complement the code generation process regarding their specific component requirements. Therefore, an adequate metamodel is generated out of the model from the first phase so that the specified capability types and capabilities are integrated as independent types into the metamodel. This generated metamodel can then be used to instantiate models for the second phase using the capabilities defined in the previous phase. The generic base structure of the metamodel can be seen in figure 5. Figure 6 on the other hand shows the metamodel generated from the example model of the first phase (figure 4). In the generated metamodel the relationship between the input model of the first phase is easily noticeable - objects are becoming classes. These classes are now available for the component producers to describe their components. The component producers are able to specify unchangeable values for attributes (e.g. *Architecture* of *CPU*), restrict the allowed values of attributes further (e.g. limit the transmission *Speed* of *Ethernet* to values supported by the hardware component), define default values for attributes (e.g. *Ethernet IPAddress*) or leave the configuration open for the following phase.

Figure 7 depicts an extract of the projects second phase model. The example model describes three hardware modules: a PC based on a dual-core processor, a controller using an ATmega 8 and an Ethernet card. Some of the capability properties are already fixed in this model (e.g. attributes of capability *CPU* of module *PC*). Others are further restricted like *Speed* of *Ethernet*. Based on the information of this modeling step, component producers are able to specify the

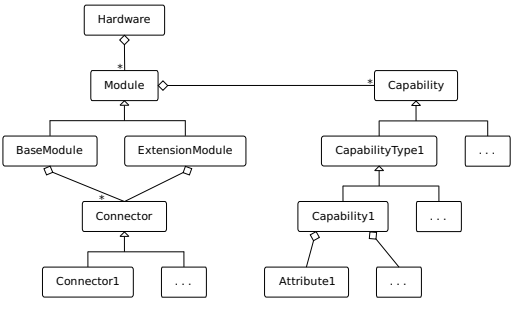

Fig. 5. Generic Metamodel of the Second Phase Based on Modules (Base and Extension Modules) and Capabilities with their Attributes

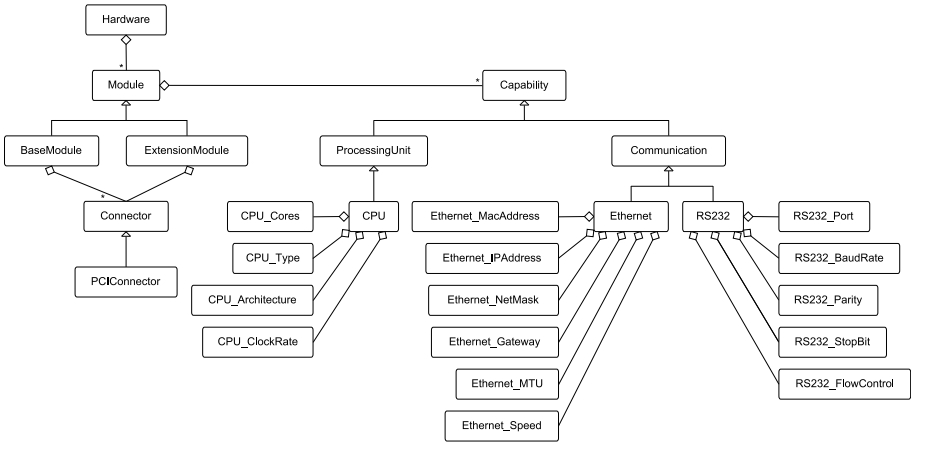

Fig. 6. Concrete Metamodel of the Second Phase Resulting from the Example Model of the Previous Phase (Figure 4)

```
<Hardware>
  <BaseModule name="PC">
    <CPU>
      <CPU_Cores value="2" changeable="false"/>
      <CPU_Type value="E6550" changeable="false"/>
      <CPU_Architecture value="x86" changeable="false"/>
      <CPU_ClockRate value="2327566" changeable="false"/>
    </CPU>
    <RS232 optional="true">
      <RS232_Port value="COM1" changeable="false"/>
      <RS232_BaudRate value="B115200" supportedValues="B9600, B19200, B38400, B115200" changeable="true"/>
      <RS232_Parity value="None" supportedValues="None, Even_Parity, Odd_Parity" changeable="true"/>
      <RS232_StopBit value="S_1" supportedValues="S_1, S_1_5, S_2" changeable="true"/>
      <RS232_FlowControl value="None" supportedValues="None, Hardware, XON_XOFF" changeable="true"/>
    </RS232>
    <PCIConnector/>
  </BaseModule>
  <BaseModule name="Controller">
    <CPU>
      <CPU_Cores value="1" changeable="false"/>
      <CPU_Type value="ATmega8" changeable="false"/>
      <CPU_Architecture value="AVR" changeable="false"/>
      <CPU_ClockRate value="8000000" supportedValues="1000000, 2000000, 4000000, 8000000" changeable="true"/>
    </CPU>
    <RS232 optional="true">
      <RS232_Port value="UART1" changeable="false"/>
      <RS232_BaudRate value="B38400" supportedValues="B9600, B19200, B38400, B115200" changeable="true"/>
      <RS232_Parity value="None" changeable="false"/>
      <RS232_StopBit value="S_1" changeable="false"/>
      <RS232_FlowControl value="None" changeable="false"/>
    </RS232>
  </BaseModule>
  <ExtensionModule name="EthernetCard">
    <Ethernet>
      <Ethernet_MacAddress value="00:00:00:00:00:00" changeable="true"/>
      <Ethernet_IPAddress value="192.168.0.1" changeable="true"/>
      <Ethernet_NetMask value="192.168.0.1 changeable="true""/>
      <Ethernet_Gateway value="192.168.0.1" changeable="true"/>
      <Ethernet_MTU value="1024" changeable="true"/>
      <Ethernet_Speed" Value="Mbps100" supportedValues="Mbps10, Mbps100" changeable="true"/>
    </Ethernet>
    <PCIConnector/>
  </ExtensionModule>
</Hardware>
```

Fig. 7. Example Model of the Second Phase Defining and Preconfiguring a PC, a Controller and an Ethernet Card

component specific code generation comprising for example the required device drivers for an actuator or sensor component.

The metamodel created in the second phase is going to be changed more often than the one of the first phase, due to the high variety of different components. These changes do not concern already existing components and therefore do not affect established models based on former metamodels, which are remaining compatible with the new metamodel as long as no components are removed or changed. Another advantage of this approach is the possibility to encapsulate the knowledge used for the component code generation. Thereby the intellectual property of the component producers is protected against others[2].

The presented approach has also been implemented for the available software components, where service instances can be created and configured.

3.3 Creation of the Application through Selection of Components

During the third phase the application developer selects and connects the components used for realizing the application like in a modular construction system. All the components defined by component producers in the second phase can be used as building blocks. They are defined in the metamodel, which is the output of a M2MM transformation based on the previous model. After selecting the appropriate components, they must be configured by the application developers according to the application requirements. The definition of attribute values and the restriction of allowed values for attributes in the second phase lead to a much simpler final configuration step and help to prevent erroneous configurations. The generic structure of the metamodel and a metamodel based on the example model of the second phase (figure 7) are presented in figure 8 and 9.

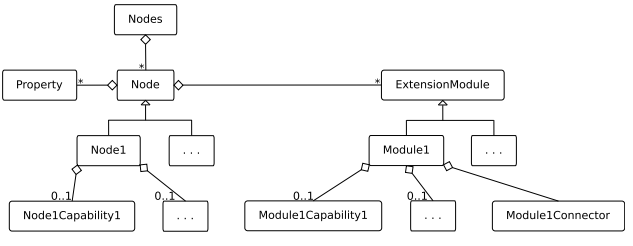

Fig. 8. Generic Metamodel of the Third Phase Based on Nodes and Extension Modules with their Associated Capabilities

As shown in figure 9, the metamodel of the third phase introduces a big increase in the number of different capabilities, which only represent specializations of the capabilities already available in phase two, e.g. *PC_RS232* and *Controller_RS232*. This is necessary to reflect the value ranges of particular component attributes of the different components produced by different manufacturers and helps to ease the correct configuration of components with respect to

[2] The support of the encapsulation is currently not implemented.

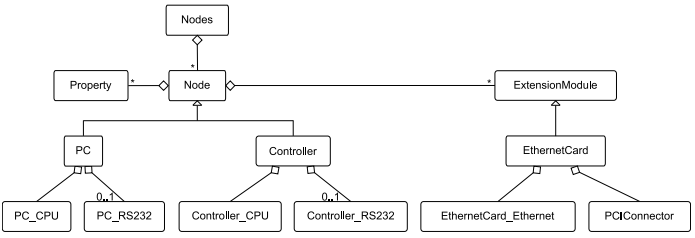

Fig. 9. Concrete Metamodel of the Third Phase Resulting from the Example Model of the Previous Phase (Figure 7)

```
<Nodes>
  <PC name="Node1>
    <PC_CPU Cores="2" Type="E6550" Architecture="x86" ClockRate="2327566"/>
    <PC_RS232 PORT="COM1" BaudRate="B38400" Parity="None" StopBit="S_1" FlowControl="None"/>
    <EthernetCard>
      <EthernetCard_Ethernet MacAddress="00:00:00:00:00:01" IPAddress="192.168.0.1" NetMask="255.255.255.0" Gateway="192.168.0.1"
        MTU="1500" Speed="Mbps100"/>
      <PCIConnector/>
    </EthernetCard>
  </PC>
  <Controller name="Node2>
    <Controller_CPU Cores="1" Type="ATMega8" Architecture="AVR" ClockRate="8000000"/>
    <Controller_RS232 PORT="UART1" BaudRate="B38400" Parity="None" StopBit="S_1" FlowControl="None"/>
  </Controller>
</Nodes>
```

Fig. 10. Example Model of the Third Phase where the Application is Assembled and Configured According to the Requirements

their offered functionality. To prevent the handling of all these auxiliary capabilities in the following code generation step it is useful to convert them back to their corresponding base types through a model-to-model (M2M) transformation before starting code generation. Thus the number of types is significantly decreased and the complexity of the code generation is reduced.

An example model with the final attribute values for the third phase taken from ϵSOA is shown in figure 10. Some of the attributes shown have already fixed values, which cannot be changed anymore. The modeled application consists of one PC module using RS232 extended with an Ethernet card and one controller using RS232.

In the following, M2M transformations and code generation are performed based on the given input from the third phase. However the mechanisms of the code generation are founded on the model of the first and second phase, which have been defined by the run-time system experts respectively the component producers. The third phase is only utilized to configure the application according to the needs.

3.4 Implementation Details

The implementation of the approach is based on the Eclipse Modeling Framework (EMF) [7]. EMF constitutes an equivalent implementation of the Essential Meta Object Facility (EMOF) [8] of the Object Management Group (OMG)[3]. MOF

[3] `http://www.omg.org/`

constitutes a standardized meta-metamodel according to the model hierarchy presented in figure 2 and specifies rich functionality for introspection. Hence it offers the possibility for generic handling of models and metamodels. The M2MM transformations are achieved through manual implementation based on the static structure of the input models: the data of the input models is read and transformed according to the tooling requirements. While the generation of the metamodel for the second phase has been straight forward, the generation of the third metamodel has been comparably complex. This is due to the fact that besides the information provided in the second model, also information from the first model has to be taken into account for the generation of the metamodel for the third phase. The model of the first phase is required, because a great part of the second phase metamodel depends on the model of the first phase. So for automatic handling of the second phase model, either the model of the first phase has to be available or the information has to be extracted from the metamodel of the second phase. In the current implementation the information of the first model is directly extracted from the metamodel of the second phase using the reflective API of EMF. This procedure prevents that incompatible models / metamodels of the previous phases are used as input for the M2MM transformation, but also requires that all the required information of the first phase model is encoded into the metamodel of the second phase even if the information is not used for modeling in phase two.

Due to the big differences between the different M2MM transformations no significant commonalities could be identified and extracted. As a consequence, the development team needs to implement almost all transformations by themselves to meet the current tooling requirements.

The current implementation only comprehends the generation of the metamodels through the M2MM transformations. The next step will be to include the model analysis and code generation of the ϵSOA project. Nevertheless a much better structure of the models could already be achieved through the separation into three different phases (capabilities supported by the run-time system, available components and application assembling). Furthermore, the extension of the models is much simpler and even possible for developers without in-depth knowledge of our development tool. Now the tool enhancement is integrated in the tool itself and requires no manual adaption of the base code, resulting in less effort when extending the tool's functionality.

3.5 Code Generation

The presented approach involves supplementary changes in the area of code generation. Coarsely the code generation can be divided into two parts as shown in figure 11. The first part comprises the generation and adaption of the components to the specified application requirements. This task has to be achieved by component producers. The second part targets the generation of the underlying run-time system, which serves as container for the components. Hence this fragment of the code generation is implemented by run-time system experts. Interoperability of the different parts is assured through the compliance

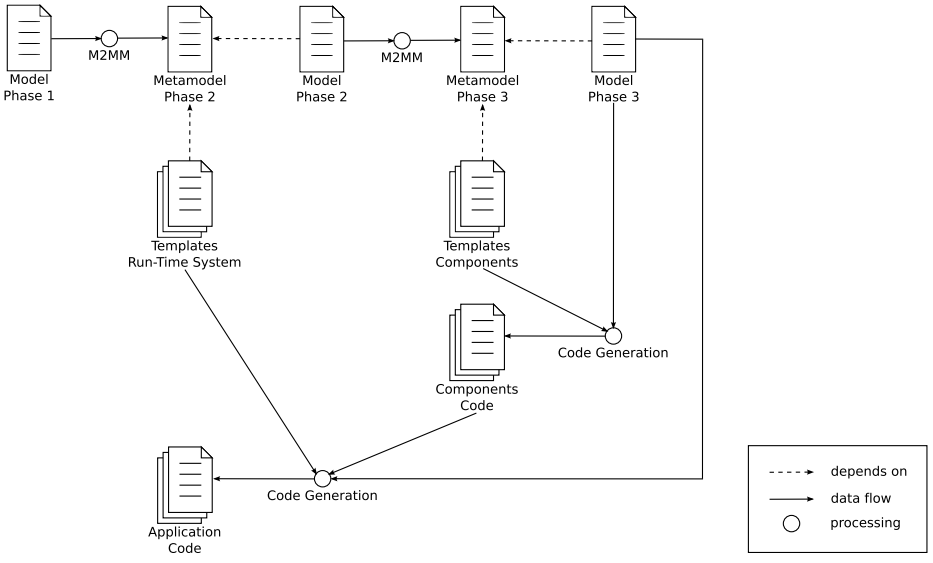

Fig. 11. Simplified Picture of Resulting Code Generation only Showing the Processing Steps for Hardware Components without any Model-to-Model Transformations

to predefined interface specifications. The responsibility for the code generation is thereby split between different groups of developers. This helps to increase the utilization of the existing expertise, as each expert must only concentrate on its own area of expertise. The system interoperability is then assured with the help of the development tool.

In the current implementation the code generation process as described above is hand written and does not automatically adapt to the inputs of the multi-phase modeling. The automated adjustment of the code generation according to the models has to be done in the future.

4 Related Work

Our multi-phase approach is based on the ideas of MDA [9] from OMG and the concept of model hierarchies [3]. The proposed approach extends MDA by introducing a new kind of model transformation. In addition to the well-known M2M transformation, where platform independent models (PIMs) are transformed towards platform specific models (PSMs), also M2MM transformations are supported. While M2M transformations operate on the same level of the model hierarchy, M2MM transformations operate across different levels of the model hierarchy. The approach introduces the possibility to extend the underlying metamodels in a flexible way. The changes to the metamodels are achieved by modifying models of a previous phase, which are the source for the M2MM transformations responsible for the creation of the metamodels. So our approach is an extension of the techniques described in MDA.

State of the art for model-driven development tools targeting component-based systems is the usage of generic libraries, as in tools like Ptolemy II [10], MATLAB/Simulink[4] and so on. The structures of those libraries are fixed and are not intended to change, meaning that the way to describe components is static and cannot be changed / extended. When a concrete component is instantiated in such systems, the structure of the component is extracted from the generic description. The components included in a model are afterwards stored in a generic way. The whole system relies upon a correct interpretation of the generic component description. In contrast, our approach supports frequent changes of the metamodel structure. For each new component type the metamodel used by the application developer is automatically adapted to perfectly reflect the component description. The result is a strong typed system with all its benefits. A drawback of this strong typing is that it is harder to use generic editors.

Bragança and Machado [11] describe a similar approach supporting multiphase modeling. In their work they use the term model promotion instead of M2MM transformation. Compared to our approach where flexibility is provided in each transformation step, they can only specialize their initial metamodel by annotating models with information utilized for automatic M2MM transformation. This restricts the power of their M2MM transformations to the predefined set of transformations offered through annotations. It also limits the usable domain concepts to the concepts introduced in their first metamodel and requires the specification of metamodel information in the model. Therefore as many M2MM transformations as needed can be conducted.

Atkinson and Kühne [12] describe problems related to shallow instantiation used in model hierarchies and suggest using deep instantiation instead. With deep instantiation, metamodels can influence the structure of all models in lower levels and not only the models exactly one level below. To use the full power of deep instantiation, related concepts need to be continuous over different levels. It is also possible to introduce new concepts on an arbitrary model level, but this requires the specification of these concepts in the base metamodel. Thus the structure of all the following levels is described in one base metamodel. Compared to our approach, where new concepts can be introduced based on the input of the previous model, their approach is restricted to object instantiation only.

The presented approach also relates to type theory / hierarchies [13] as it establishes its own kind of type hierarchy. The advantages of the proposed approach in contrast of using ordinary type hierarchies of object oriented systems become obvious with respect to refinement of relations. In object oriented systems, a refinement of relations can only be done in addition to the inheritance of the original relation. The suggested approach provides transformation-based rules to support the refinement in a natural way: the generic relation of components is removed and replaced by component-specific relations. In addition, the interface to access generic relations can be automatically generated. Therefore, the approach helps to cope with the potential danger of inconsistencies when modifying objects using both, the generic and the specific, interface.

[4] `http://www.mathworks.com`

Another interesting approach to extend metamodels in a flexible way is presented by Morin et al. [14]. They use aspect-oriented modeling (AOM) techniques to weave variability into metamodels. A similar technique could be used to extend our metamodels, but inevitably results in a separate aspect model for each new component instead of having a central point where all the components are defined. Furthermore, the extension of the system always requires a manual aspect model creation. As the user is able to influence the whole existing metamodel with his aspect model this can lead to unnecessary failures. In addition, it is unclear how to adapt this approach to support multi-phase modeling.

5 Conclusion

Domain-specific model-driven tools and component-based approaches are used to simplify the development process of embedded systems. Tools encapsulate system commonalities as domain concepts in the modeling language and in the code generation. Components are used to encapsulate frequently used functionality. Up to now, these concepts are mainly used for the development of applications, but not for the metamodels used by the development tool itself. Due to the heterogeneity of embedded systems, this focus is problematic, since the metamodels must be frequently changed if new types of components are introduced.

In this paper, a new approach for a multi-phase model-driven development of component-based systems has been presented based on model-to-metamodel (M2MM) transformations. Additionally to the support of the application developers, this approach also focuses on the support of the run-time system experts and component producers. In individual phases, these experts can focus on their view on the system and extend the system easily. In context of ϵSOA, it has been shown that our approach presented in this paper is well suited for the new use case and offers a significant improvement for the development of component-based systems. Resulting advantages are a better structure of the metamodels and a separation of the code generator development into two separate phases (run-time system and components).

Our approach has been demonstrated for modeling in the context of a development tool for distributed sensor / actuator systems. The current results indicate a much better extensibility of the whole system and a faster and easier development process. In the future the code generation of the tool has to be adapted to this approach and finally evaluated with respect to already existing systems.

References

1. Szyperski, C.: Component Software. Addison-Wesley Professional, Reading (November 2002)
2. OMG: Common Object Request Broker Architecture (CORBA/IIOP) Specification Version 3.1. (January 2008)
3. Stahl, T., Voelter, M.: Model-Driven Software Development: Technology, Engineering, Management, 1st edn. Wiley, Chichester (2006)

4. Buckl, C., Sommer, S., Scholz, A., Knoll, A., Kemper, A.: Generating a tailored middleware for wireless sensor network applications. In: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, pp. 162–169. IEEE, Los Alamitos (June 2008)
5. Buckl, C., Sommer, S., Scholz, A., Knoll, A., Kemper, A., Heuer, J., Schmitt, A.: Services to the field: An approach for resource constrained sensor/actor networks. In: 2009 International Conference on Advanced Information Networking and Applications Workshops, pp. 476–481. IEEE, Los Alamitos (2009)
6. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
7. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008)
8. OMG: Meta Object Facility (MOF) Core Specification Version 2.0. (January 2006)
9. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. (June 2003)
10. Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H.: Heterogeneous concurrent modeling and design in Java, vol. 1, Introduction to Ptolemy II. Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley (April 2008)
11. Bragança, A., Machado, R.J.: Transformation patterns for multi-staged model driven software development. In: SPLC 2008: Proceedings of the 2008 12th International Software Product Line Conference, Washington, DC, USA, pp. 329–338. IEEE Computer Society, Los Alamitos (2008)
12. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 19–33. Springer, Heidelberg (2001)
13. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993)
14. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., Jézéquel, J.M.: Weaving variability into domain metamodels. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 690–705. Springer, Heidelberg (2009)