FROM MODEL TO REQUIREMENTS: PATTERN-BASED ANALYSIS IN DISTRIBUTED DEVELOPMENT OF EMBEDDED SYSTEMS

Clive Thomsen, Judita Kruse, Rolf Ernst
*Institute of Computer and Communication Network Engineering,
Technical University of Braunschweig*
thomsen|kruse|ernst@ida.ing.tu-bs.de

Abstract: As design complexity of embedded systems raises with each new technology generation, development becomes a more and more challenging task for a single manufacturer or system house. To counter this, we see a continuing trend to spread development over several companies with differing core competencies. This is either done by purchasing ready made IP blocks or by setting up a distributed design process. For both approaches, the system designer has to designate a sufficient specification for the components externally supplied, to ensure proper functionality after system integration.

To support the designer in this task, we propose pattern-based analysis of a central structural system model. We present an approach to derive inter-system dependencies of design properties by model transformation, which allows the efficient use of analysis and verification tools in early design stages. This enables easy specification validation.

Keywords: Requirements, Model Transformation, Pattern-based Analysis, Embedded Systems

1. INTRODUCTION

Embedded systems are getting more and more complex. In a single car, a dozen of microprocessors can be found today, spread over several control units, communicating over a variety of networks and field buses. With the increasing use of MpSoCs we expect a dramatic boost in complexity, as more and more service functions and features can be added to the systems while additionally allowing lower production cost through higher integration level.

This poses a high challenge on the development process. Often, the development is still based on text documents and manual data exchange. New ways and tools are needed to face the complexity explosion.

Let's have a look at the following setting: A *customer* gives a *system house* the assignment to develop an embedded system. The customer poses a wide set of functional requirements on the unit, specifying unit operation and protocol of external interfaces. Solutions for generally dealing with these requirements can be found in [3],[4].

Additionally there will be a set of requirements not directly coupled with the unit's function. These non-functional requirements can be roughly grouped into four categories:

Development effort: The development may not cost more or take longer than the customer admits.

Resource usage: How many resources like power or memory are used by the unit.

Unit performance: The rating of the unit. Often given as throughput, response time or real-time requirements.

Operating conditions: The unit must be compliant with environmental influences like radiation or extreme temperatures.

At the system house a *system architect* will try to find a *system structure* which provides the requested functionality and is suitable in terms of the non-functional requirements. If very high performance or low power consumption is needed, the architect may choose an implementation in specialized hardware. If the architect focuses on platform reuse, he will favor a co-design approach, where the functionality is implemented as software, running on a ready-made embedded processor.

Next, the architect will specify sets of requirements for each sub-system. These sub-requirements must fulfill two prerequisites:

- *Assertability:* The requirements must be assertable by the sub-system suppliers. This can be a problem if the requirements are technically challenging, as the supplier may refuse to guarantee meeting those on a contractual basis. We have suggested flexible contracting schemes as a solution to this issue in [5],[6].
- *Validity:* The requirements on the sub-systems must be valid in the overall system context. The architect must ensure, that meeting all sub-requirements is sufficient for meeting the overall requirements.

To do so, the system architect needs knowledge about the dependencies between different system properties. As shown in figure 1 each property of the overall system is dependent on the properties of the sub-systems. The actual characteristics of these dependencies D_1 to D_4 is determined by the system structure the architect has defined.

The system architect has two choices to deal with these dependencies. She may rely on her design experience to roughly estimate them and to determine sub-requirements which will lead – to her best knowledge – to satisfactory

Figure 1. Dependencies between sub and overall system

results. However, in complex systems, there is an imminent risk of missing important dependencies. Additionally, when only estimating dependencies, the designer tends to be conservative, to lower her risk of missing overall requirements, leading to an over-sized design.

For very challenging or tight requirements, she may want to use a corresponding analysis tool kit. Especially in the area of real-time and throughput many promising approaches are currently under development or becoming available [7],[8],[9]. These tools have in common, that the architect has to provide the system structure as a specific input model for the tool. For complex systems this modeling may be a time-consuming, cumbersome and thus error-prone task.

Both approaches are insufficient for today's complex embedded systems. As a solution we want to suggest pattern-based analysis of a structural design model. This approach is based on the observation, that the architecture of complex systems can be decomposed into a limited set of simpler structural entities, so called "patterns". A pattern is the encapsulation of a well approved design scheme [1],[2]. Based on this observation, we want to achieve an automated transformation of the system architecture into models suitable for analysis tools: The system architecture is scanned for known design patterns, these are transformed into the corresponding analysis patterns and composed into an overall analysis model.

We will introduce our approach of pattern-based analysis for embedded systems in the next section. After presenting an example in section 3 we will conclude in 4.

2. PATTERN-BASED ANALYSIS

This section introduces to pattern-based analysis and how it supports design processes.

Figure 2 shows the major building blocks of our suggested system:

Figure 2. Data-flow for pattern-based analysis

The central *design property and architecture database* contains a *structure model* capturing the system architecture and properties of the different system components.

The database is accessed via the *model data interface* (MDI). This interface provides a generic view on the design structure, decoupled from the modeling language used in the database. Further details can be found in section 2.1.

The *model generator* uses the MDI to transform the structure model found in the database into an analysis model targeted for a certain analysis. This model is augmented with all needed input values from the property database and processed by the analysis tool. The obtained results are back annotated into the database, using the MDI again.

Often, one analysis depends on results obtained by other tools. We monitor the property database and re-execute tools, if their input values have changed. This builds up a network of tools and ensures consistency within the database.

The system architect can now enter different the cornercases scenarios of her specification into the property database, to test if these will lead to satisfactory overall system behaviour. Of course, this procedure could be automated, allowing convenient specification validation.

For conducting the model transformations, we have developed the *model traverse language* (MTL), an XML-based script language described in section 2.2.

Often the design architecture is composed of user-defined components unknown to the model generator. The *type mapping* provides equivalence relations between user-defined types and those known to the model generator.

2.1 The Model Data Interface

In this section we will have a closer look on MDI, a generic interface to access a system architecture model.

To automatically derive dependencies from a system model, we need information about the internal communication paths and the mapping of functionality onto implementation resources. Communication is especially important for performance requirements like throughput or response time. To calculate these

properties we need knowledge of all components of a communication path and their behavior. The mapping determines how resources are shared between different system components. To calculate to load of a CPU for example, we need to know which tasks should run on it.

In today's requirement management tools, hierarchical models are used, similar to the AP233 data model [10]. These simple models are insufficient for an automated dependency analysis, as they neither provide information about communication nor about mapping. A new, upcoming modeling standard dealing with these issues is SysML, an UML extension [11]. But SysML is still in the standardization phase and no modeling tools are available yet.

To overcome this situation, we have defined the model data interface, a generic representation of structure models. Figure 3 shows an UML diagram

Figure 3. UML description of MDI

of the interface.

The core class of the interface is the *component*. A component represents an entity in the design to be modeled. An arbitrary name can be assigned to each component. As today's designs are mostly organized hierarchically, component can be the composition of further child components, allowing the modeling of a component tree.

To model communication, a component provides a set of *ports*, which can be interconnected to form channels. We distinguish *origin* and *destination* port classes to reflect the flow of the modeled communication. For example, origin ports correspond to bus masters, destination ports to bus slaves.

To capture mappings, the modeling language provides *associations*, a labeled and directed relation between components.

Components, associations and ports share the super class *Node*. Each node owns an ID to uniquely identify the modeled entity in the system. Additionally

a node allows storage and retrieval of properties using textual key values. As data format for properties XML DOM elements are used [12].

Ports and components carry a "type" attribute to denote the characteristics of the underlying entity. In system models, very precise and specialized types are used, like "ARM" or "CAN", while analysis models often use more generalized types, like "CPU resource" or "bus". The "IsKindOf" evaluates the type mapping to check, if the port or component is the specialization of a generalized type.

The structure model can be separated into different *layers*. A layer is a sub-model presenting a certain view on the system. For example, typical layers for embedded systems would be "hardware architecture", "software architecture" or "data flow".

2.2 Analytical Patterns in MTL

This section features a brief discussion of the model traverse language (MTL). A MTL script describes how to generate an input model for a certain analysis tool targeted by the script. MTL is an XML-based script language, interpreted in a two stage process.

In the first stage, a *pattern processor* interprets the MTL script to traverse the structure model, inspecting the different components encountered to find known design patterns. When a pattern has been recognized, the corresponding analytical model is generated in an *intermediate* output format.

This intermediate contains the generated model encoded as XML and a description of its property dependencies. The dependency description is used to re-invoke the corresponding analytical tool, when component properties change, to ensure database consistency. An example of an intermediate output can be found in figure 8.

The second stage is responsible for tool invocation. First all needed component properties are fetched from the database and annotated in the intermediate. Next a wrapper script is called to invoke the analytical tool and to extract the newly calculated component properties from the tool output.

The pattern processor construes the structure model as *model graph*, using components, ports and association classes as node set and their relations as edge set. MTL statements operate on the model graph and can be grouped in three categories.

Traverse statements allow to navigate along specified edges of the model graph. These statements can have child-statements and a *filter expression*. A filter expression allows to select on which nodes the child-statement shall be executed, based on node properties like its type.

Control-flow statements are sub-function calls or repeat loops, for example. A special tag statement and a corresponding filter expression allow setting and

testing of arbitrary flags on model nodes. Thus algorithms depending on graph search can be implemented.

Generate statements construct the intermediate representation. A node statement inserts specified XML elements into the intermediate representation. Data statements indicate that a property should be accessed for all members of the node set. This is done by inserting special markers denoting the identifier of the model node and the name of the accessed property.

3. EXAMPLE

In this section we look at a small design to exemplify our approach.

The system to develop is a position controller. The unit must keep track of its position and can make corrections through a PWM controlled motor. The unit shall exchange target and actual position value over a network interface with an external system. Additionally, the unit should control the motor's temperature. As platform, a microprocessor with CAN Interface is used, The data exchange and the regulator itself shall be implemented in software, while for temperature control a threshold switch is used

After modeling the system, a SymTA/S [9]analysis shall be performed to evaluate real-time performance, followed by a simple power estimation.

We will separate our example into three layers: A functional layer, containing all functional units and their data-flow, a software and a hardware layer. Figure 4 shows the functional layer. As the complexity of the design is very

Figure 4. Functional view and software architecture of example system

low, we get along with a limited set of types: "Sensors" and "actors" as standard components of controllers, "processes" providing the functionality, as well as "input" and "output" ports to communicate with the environment.

The software uses a "scheduler" to manage two "tasks", which implement the functionality of the processes found in the functional model (also 4).

In figure 5 we see the hardware layer of the structure model. All communication is handled by a central "packet bus". We find the implementations

Figure 5. View on example platform

of the sensors and the external interface acting as bus masters, as well as the CPU-core. The actors are connected as slaves devices.

Figure 6 shows the type mapping used for the hardware layer. For the power

Figure 6. Graph of types in hardware layer

analysis model knows about "static consumer" and "dynamic consumer" type. A static consumer shows a nearly constant power usage, while the dynamic consumer's power usage is linearly dependent on its utilization. During model generation, bus and CPU will be mapped to a dynamic consumer, while for all other elements static power usage is assumed.

Not shown explicitly shown in figures 4 and 5 is a "schedules" association connecting scheduler and CPU, as well as "implemented-by" associations between layers, which are indicated by name equality of components. Also, as the bus is the only communication device in the system, we can spare the mapping of communication channels.

To generate the SymTA/S model, a corresponding analysis pattern is applied on the functional view. The pattern generates appropriate nodes for all found components and special communication nodes to map the channels. The

the software layer will be evaluated to associate the two nodes created for the Control and the Monitor process with a CPU resources.

Now the needed input data can be fetched from the database, like execution times of the tasks, length of messages, communication behavior of the sensors, or clock rate of bus and processor, etc.. A wrapper script now feeds model and data into SymTA/S. Figure 7 shows a screen-shot of the resulting SymTA/S

Figure 7. SymTA/S model of system

model.

Figure 8 is an excerpt of the intermediate model generated from the power analysis pattern. For better readability, reference identifiers have been replaced by the name of the corresponding model component. The headline denotes how the file was produced. The "dataout" tag declares that this intermediate will provide the property "power". For static power consumer, it will just fetch the power values from the property database. For dynamic power con-

```
<intermediate pattern="power" target=overall system>
  <dataout ID=overall system property="power"/>
  <static>
    <datain ID=position sensor property="power"/>
  </static>
  <static> [ ... ] </static>
  <dynamic>
    <datain ID=CPU property="idle power"/>
    <datain ID=CPU property="load power"/>
    <datain ID=CPU property="utilization" />
    <dataout ID=CPU property="power" />
  </dynamic>
  <dynamic> [ ... ] </dynamic>
</intermediate>
```

Figure 8. Output of power pattern applied to overall system

sumers, idle power, load power and utilization are fetched to compute the components consumption. When the required properties have been fetched from the database, the file can easily be evaluated directly by a corresponding script, which calculates the dynamic powers and the overall power consumption.

4. CONCLUSION

Early design validation becomes a more and more important issue in embedded system design processes. Pattern-based analysis is applied on a central model capturing the design architecture. It can be used to invoke multiple analysis tools and to track down inter-dependencies in the design. We have introduced the model data interface for accessing a user supplied structure model and MTL as a language to express pattern-based model transformations.

Our further work will be on interfacing our approach with standards for requirement handling and system modeling like AP233 and SysML. Additionally base of available patterns and supported analysis tools should be extended.

In the long run we want to study the aspect of data quality in the property database and how our system can be used for risk management in later design stages.

REFERENCES

- [1] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley Publishing Group (1995).
- [2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel. *A Pattern Language*. Oxford University Press, New York (1977)
- [3] Mueller, D. *demanda II, A prototype for Requirement Capture & Formalization*. In *IMW - Institutsmitteilung Nr. 26*, Germany, Clausthal (2001), ISSN 0947-2274.
- [4] DOORS. <http://www.telelogic.com/products/doorsers/doors/>.
- [5] J. Kruse, T. Volling, C. Thomsen, R. Ernst, T. Spengler. *Towards flexible Systems Engineering by Using Flexible Quantity Contracts*. In *Proc. Automation, Assistance and Embedded Real Time Platforms for Transportation (AAET'05)*, Braunschweig, Germany, 2005.
- [6] J. Kruse, T. Volling, C. Thomsen, R. Ernst, T. Spengler. *Introducing Flexible quantity contracts into distributed SoC and embedded system design processes*. In *Proc. Design, Automation, and Test in Europe (DATE'05)*, Munich, Germany, 2005.
- [7] S. K. S. Chakraborty, L. Thiehle. *A general framework for analyzing system properties in platform-based embedded system designs*. In *Proc. Design, Automation and Test in Europe (DATE'03)*. Germany, Munich, Mar. 2003.
- [8] AADL. <http://www.aadl.info/>
- [9] SymTA/S. <http://www.symta.org>
- [10] AP233, <http://ap233.eurostep.com/>.
- [11] SysML, <http://www.sysml.org/>.
- [12] *Document Object Core Model*, <http://www.w3.org/TR/DOM-Level-2-Core/core.html>.