J. Lilius, J. Lindqvist, I. Porres, D. Truscan |
T. Eriksson, A. Latva-Aho, J. Rakkola

Testable Specifications of NoTA-based Modular Embedded Systems

Turku Centre *for* Computer Science

TUCS Technical Report
No 841, September 2007

Testable Specifications of NoTA-based Modular Embedded Systems

J. Lilius, J. Lindqvist, I. Porres, D. Truscan
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5, 20520 Turku, Finland
{Johan.Lilius, Johan.Lindqvist, Ivan.Porres, Dragos.Truscan}@abo.fi

T. Eriksson, A. Latva-Aho, J. Rakkola
Nokia Research Center
P.O.Box 407, 00045, Helsinki, Finland
{Timo.Eriksson, Antti.Latva-Aho, Juha.Rakkola}@nokia.com

TUCS Technical Report
No 841, September 2007

Abstract

We propose an approach to specifying embedded devices based on the Network on Terminal Architecture (NoTA) and show how it allows the specification of NoTA components, including service interfaces, and timing and energy consumption constraints. The main purposes of such specifications are to enable vendors to provide already tested component implementations with respect to specifications, and system designers to test these components in integration. The proposed specifications feature both a graphical notation for facilitating the specification process using dedicated tools and a textual one for exchanging component specifications between system designers and vendors.

Keywords: NoTA architecture, testing specifications, SOA, SOC

TUCS Laboratory
Embedded Systems Laboratory
and
Software Construction Laboratory

1 Introduction

Due to increasing complexity of current system specifications, system designers need to focus on integrating the functionality of different components into a meaningful product, rather than taking care of the low-level implementation details of different components. As such, more and more parts of current embedded devices have to be leveraged to subcontractors or third-party vendors. However, drawing the full benefits from this approach requires the specification framework to overcome several challenges.

Firstly, one has to utilize specifications that facilitate the communication between the system designer (integrator) and vendors, by clearly specifying the characteristics of components to vendors. The lack of such specifications and clear modularity has lead to a situation where potentially crippling problems, such as the combined power consumption of two components exceeding the capacity of the power source when running a certain test case, are detectable only after system integration.

Secondly, the specifications of the components provided to vendors have to be accompanied by a set of tests which enable the vendors to test the implementations of their components before delivering them to the system designer. This will leverage off from the designer's shoulders the testing of component implementations and thus will reduce the number of re-spins of component implementations for fixing errors.

Thirdly, not only component specifications have to be testable, but also the system as a whole, in order to enable the designer to test different components in integration in the final product. This can be accomplished by defining system-level testing scenarios that span several components.

Lastly, reuse on different levels has to be employed in order to speed up the design process and reduce the time-to-market of new products. Having well defined specifications of existing components and benefiting from organized specification and component libraries, system designers should be able to develop new products in a relatively short time-frame.

In order to address some of these issues, the *Network on Terminal Architecture* (NoTA) [1, 2] has being developed (at Company Name Removed) as an attempt to enable systems developers to create highly modular, easily integrated embedded systems, using a testable specification format. NoTA employs the concepts behind Service-Oriented Architectures (SOA) [3] for development of embedded devices, by dividing an embedded system into a set of components (called *subsystems*), each implementing a set of *services*. The architecture promotes the creation of loosely coupled components implementing well-defined interface specifications and communicating using standardized protocols.

In this paper we propose an approach to specifying NoTA-based systems which attempts to clearly specify to the vendors the required characteristics (service interface) of a given subsystem along with specifications for testing the subsystem

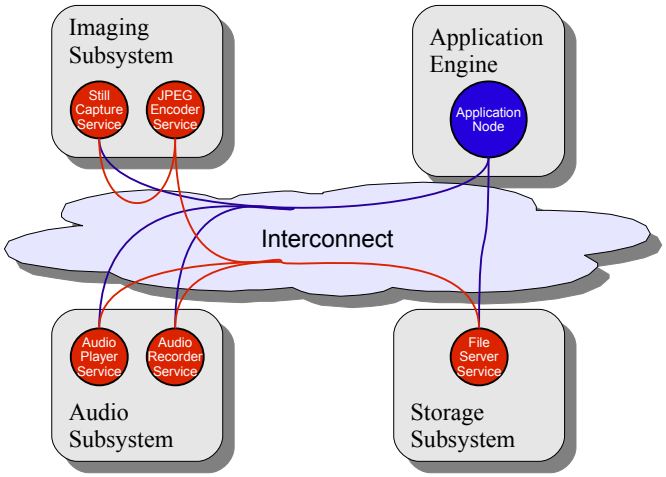

Figure 1: Example NoTA device structure

against the requirements (such as timing and energy consumption test specifications). The proposed specifications are represented using both a graphical notation for facilitating the specification process in dedicated tools and a textual one for enabling the exchange of specifications between designer and vendors.

The paper continues as follows: the next section introduces the main concepts behind the NoTA architecture. Section 3 delves into presenting how different aspects of NoTA, such as requirements and testing specifications, are specified. We accompany our presentation with excerpts from a mobile terminal case study. Section 4 discusses the tool support for editing the specifications and how the graphical specifications can be transformed into a textual format. We conclude with final remarks.

2 The NoTA Architecture

A NoTA system consists of a number of *service nodes* and a number of *application nodes* communicating through the NoTA *Interconnect*. The service and application nodes are distributed on a number of modules denoted *subsystems*.

2.1 NoTA Subsystems

Each subsystem is conceptually an independent entity communicating with the rest of the system only through the Interconnect. Physically, a subsystem may reside completely on its own board, containing power regulators, processors etc. of its own, it may be integrated with other subsystems on-chip, or anything in

between. However, regardless of the physical level of integration the architecture still remains modular in the sense that the communication between subsystems is conducted through the Interconnect in exactly the same way from the point of view of the software running on top of the subsystem. An anticipated benefit from this is that new features may be introduced into pioneering low-volume devices as subsystems physically constructed using off-the-shelf components and later be incorporated on-chip in higher-volume versions of the devices without changing the architecture of the system. This freedom to choose the level of integration is seen as a key benefit brought by the NoTA approach [2].

The application nodes constitute the interface toward the user of the system and request functionality from the service nodes as needed. The service nodes themselves may use functionality supplied by other services in order to deliver what is requested of them. In the example system in Figure 1, this is the case with the *Still Capture* which uses the *JPEG Encoder* to encode raw images into the JPEG format. These images may then in turn be stored on the *File Server* before a handle to the stored image is returned to the *Application Node*. Looking at the example system, it is interesting to note that a high-end system may have exactly the same structure as an entry-level system, only the service implementations might differ.

Services are defined using a *Service Interface Specification* (SIS). Upon service "power-up", the service registers itself with the Interconnect using a service ID associated with the interface specification. Other services may then request a connection to a service conforming to that interface specification by using the ID, whereupon the Interconnect sets up a connection channel. In this way, the Interconnect is the only entity aware of the physical location of the services.

If several services conforming to the same interface specification are registered on the system, the Interconnect chooses to which of these service instances the node requesting the connection is connected. The choice may be made purely based on the structure of the network lying between the nodes, or on other factors. At present, no rules have been defined for how this choice should be made, and it is thus up to the Interconnect implementer to make sure the choice is made intelligently.

2.2 NoTA Interconnect

The NoTA Interconnect provides the communication capabilities needed for the communication specified in the service interface specifications. The communication is asynchronous and in general the Interconnect guarantees only that it makes its best effort to deliver messages. In other words, if stronger guarantees are required, they may be obtained using additional logic inside the subsystems.

The Interconnect provides two distinct communication patterns – asynchronous messaging and data streaming. Typically, the first is used for control commands while the second is used whenever large quantities of data are transferred between

subsystems. Data streaming is accomplished over the NoTA Interconnect using the *Direct Object Access* (DOA) protocol.

The interface for communicating with a service node over the Interconnect is defined by the SIS implemented by the service node. This will be discussed in more detail in the following section.

3 NoTA Subsystem Specification

A NoTA system is typically designed in a top-down manner. Starting from the *requirements* of the system, the designer identifies units of closely related functionality (the *services*) that the system should provide in order to satisfy the requirements. By applying different usage scenarios (extracted from the requirements) the identified set of services is partitioned into subsystems. The partitioning process heavily depends on designer's experience and on business reasons. The subsystem specifications are then distributed to vendors to be implemented in silicon.

A subsystem specification comprises two parts: (1) a list of services (and their SIS) that the subsystem should implement and (2) a set of usage scenarios (use cases) that the subsystem should support.

The NoTA specifications benefit from a dual representation, having both a graphical and a textual format. The graphical specification language has been constructed using metamodeling techniques (i.e., by defining a modeling language in the form of a metamodel), being intended to facilitate NoTA specifications in graphical modeling environments. The textual specification language is defined by extending the Web Service Description Language (WSDL), the most popular standard defining Web services, and also the standard that enables service-oriented principles to be realized in practice in Web services. WSDL is defined as an XML Schema [7], and thus service descriptions built using WSDL have the form of XML documents adhering to the WSDL XSD (XML Schema Definition) [8]. Nevertheless, although the two representations of the NoTA subsystem specification are defined using different techniques, they are basically equivalent. This enables us to translate graphical representations into textual ones and vice-versa at any time without losing information.

In the following, we will briefly present the different aspects of NoTA subsystem specifications using the graphical specification language. The main focus of this presentation is on how the language can be used to specify NoTA subsystems and their services, rather then how the language has been defined. We defer for Section 4 the exemplification of the textual specification form.

We exemplify the approach with excerpts from a mobile terminal device case study. As a result of the service identification process several services have been identified, as follows: *StillCapture* for capturing still images in bitmap (BMP) format, *JPEGEncoder* for encoding bitmap images into JPEG format, *APlayer*

for playing MP3 encoded data streams, and a *FileServer* service for storing the resulting image files or for providing MP3 data streams. A possible partitioning of these services into subsystems is shown in Figure 2.

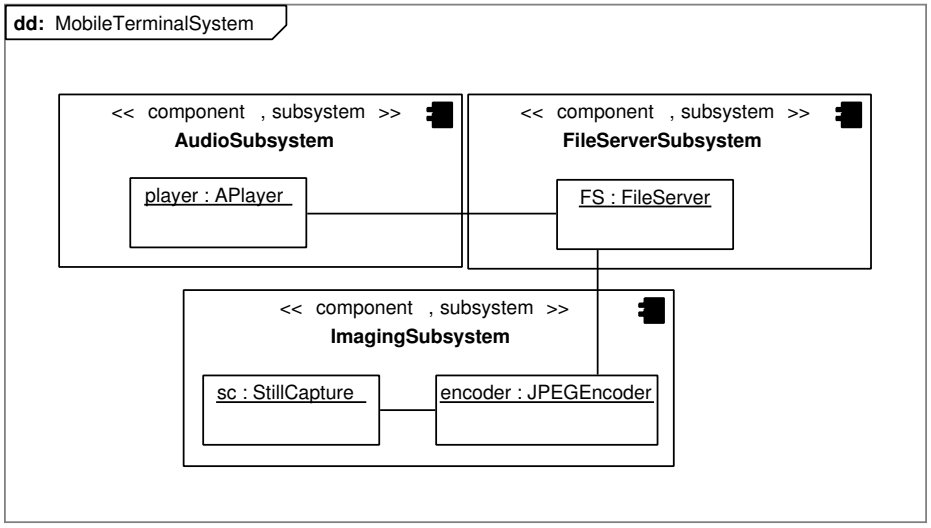

Figure 2: Possible partitioning of the mobile terminal system under study

3.1 Service List Specification

The interface for communicating with a service node is defined by the SIS, which is a central concept in the specification and testing of NoTA-based systems. The SIS of a NoTA service consists of two main parts: a control interface and a data interface.

3.1.1 Control Interface

The *control interface* depicts the part of the SIS that allows the invocation of different functionalities of a service. The control perspective is specified by two artifacts: *Interface Specification* – a list of input/output messages and their associated parameters that the service can send and receive, and *Behavior Specification* – a protocol state machine depicting the externally observable states of the service along with the messages that can be received or sent by the service in each state. In addition, the behavior specification depicts the messages sent by a service to invoke (use) functionality provided by other services.

As an example, we show in Figure 3 the interface specification of a *StillCapture* service, while in Figure 4 we present the behavior specification of the same service.

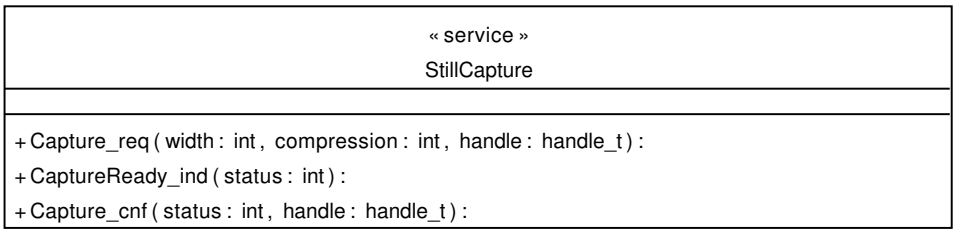

Figure 3: Interface specification of the *StillCapture* service

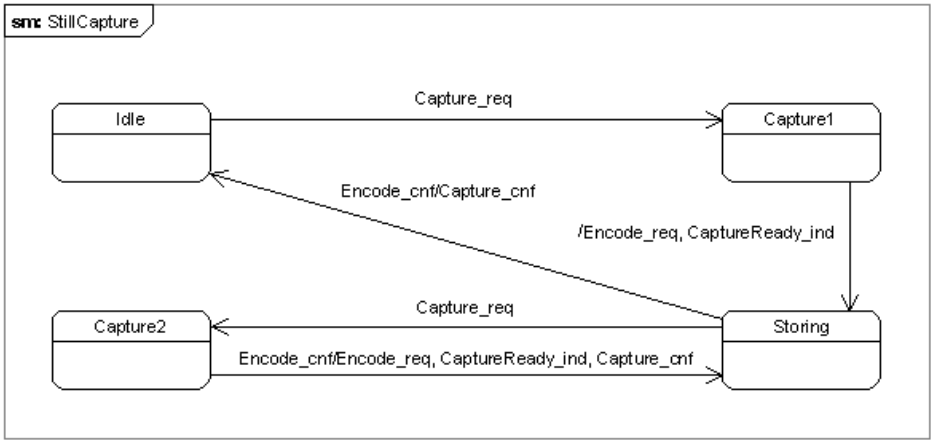

Figure 4: Protocol state machine of the *StillCapture* service

3.1.2 Data Interface

The *data interface* specifies the types of data, namely MIME types [5], a service supports for communicating with other services using the DOA protocol. Thus, each service is accompanied by a *Data Handling* specification – that is, a set of *data handling patterns* – specifying how the service is expected to communicate using each data type. Each pattern describes properties like bandwidth, latency, power consumption of the communication. Different patterns mirror the different requirements posed by different types of data communication. For instance, the performance requirements for streaming `audio/mp3` data to a player may vary greatly in terms of packet size, average bandwidth, etc. as compared to saving an `image/jpeg` on a file system.

A generic data handling model is used to represent all the data handling patterns. The handling model, also referred to as *DOA FSM*, specifies a given data handling pattern as a state machine with two states: *Idle* and *Active*. The transfer of data only takes place during the *Active* state with a specified *bandwidth*. If for a given amount of time (*timeout*) no data has been transferred, the state of the DOA FSM is changed back to *Idle* in order to save energy. The DOA FSM also models the power consumption/energy and delay of a given data handling pattern, by

specifying the *power* consumed in each state, and the *energy* and *delay* implied by changing the state of the FSM. Future work will attempt to extend this model to contain other performance characteristics, like peak and average energy for transitions. An example DOA FSM of the *APlayer* service for the `audio/mp3` MIME type is given in Figure 5.

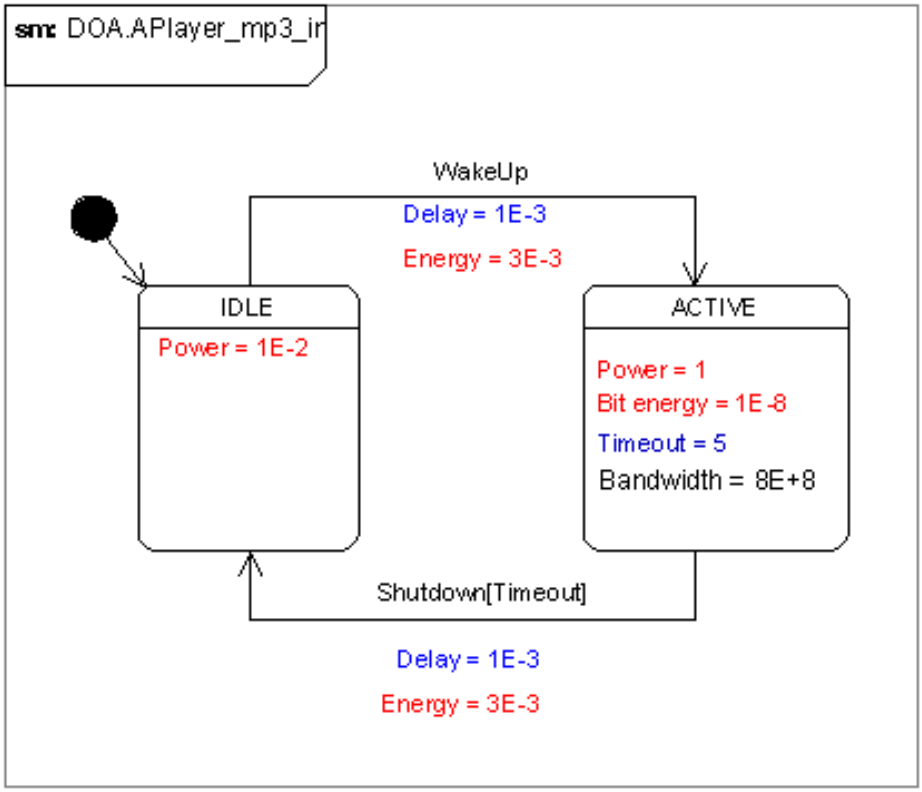

Figure 5: DOA state machine for the *audio/mp3* MIME type

The DOA FSMs can be employed to estimate the properties (e.g., timing and energy consumption) of a data transfer between two services. One of the services will be the source and the other one the sink, and each of them may use a different DOA FSM. By computing the cross-product of the source and sink DOA FSMs one obtains a state machine modeling the characteristics of a given data transfer.

3.2 Use Case Specification

In our approach we regard as the *Requirements* a collection of documents written in natural language, associated standards and specifications, as well as user stories. Starting from these requirements one identifies the usage scenarios (or *use cases*) and their interaction with the external environment of the system under design.

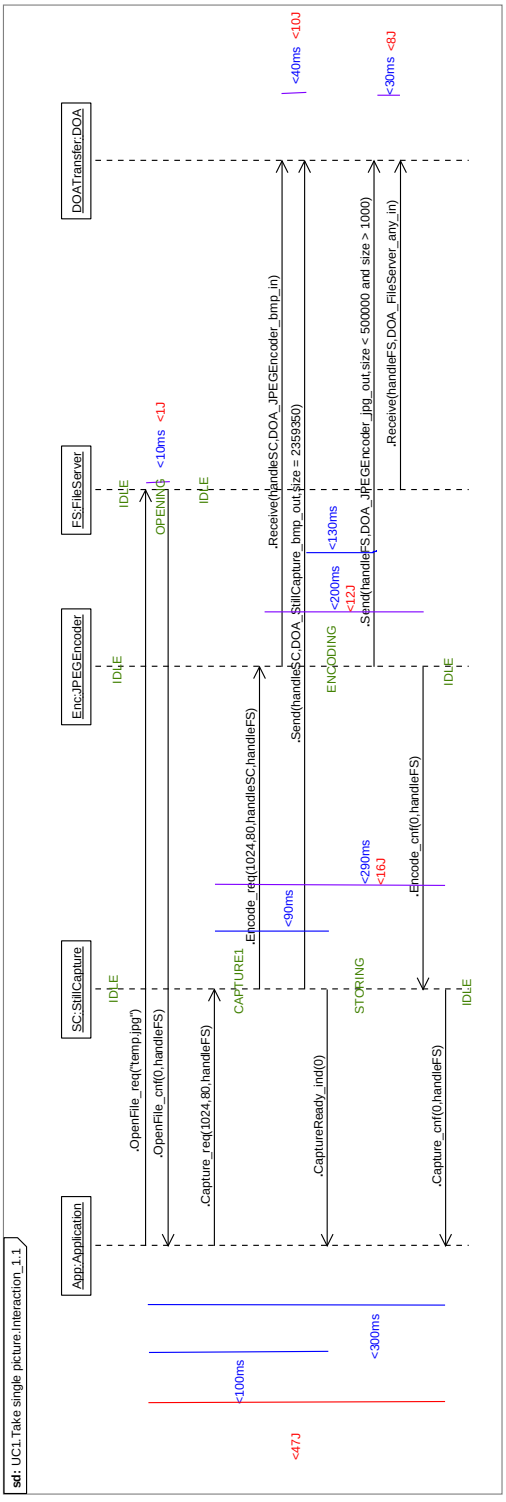

Figure 6: Interaction diagram for use case *UC1. Take single picture*

This step produces a *Use Case Model* in which each use case is accompanied by a textual description (including both functional and non-functional requirements).

A *Use Case Library* is used to provide support for reuse. The pre-defined use case specifications stored in this library allow, once a use case is added to the *Use Case Model*, to provide already built *service interactions*, *service* and *subsystem* specifications which implement that particular use case.

In the case of the *FileServer* system, we have extracted four usage scenarios that have to be supported by the system: *UC1. Take single picture*, *UC2. Take picture series*, *UC3. Play MP3 file* and *UC4. Browse file system*. Each scenario is accompanied by a description of its functionality and a number of non-functional requirements. For instance, the description of the *UC1. Take single picture* is as follows:

> Take a jpeg image with resolution 1024*768 and store it on the file server as ”temp.jpg”. From the time the image is requested to the time it is available on the file server no more than 300ms should have elapsed. The image should be captured (but not necessarily encoded or stored) within 100 ms from the time of the request. With one fully charged Li-Ion battery (1 Ah, 3.3 V) the user must be able to take 200 VGA-sized still images, including 20 % overhead for additional tasks.

Based on the services identified at the previous step and on the textual description of each scenario, we analyze the interaction between the services belonging to the same scenario. The interaction is depicted both in terms of asynchronous *messages* passed between the environment and services, as well as among services, and in terms of data transfers between services. A message may have an input or an output direction with respect to a service and may be accompanied by a list of *parameters*.

The ordering of messages received or sent by a service is depicted by their location on a *service lifeline*. In addition, a lifeline may contain several *zones*, depicting the sequencing of *observable states* of a service in time. Using zones allows us to specify in what state of a service a given message can be sent or received. In addition, it enables one to specify what the next state of a service should be after a message is received or sent. For simplicity reasons, we use the convention that a message is received or sent only at the end of (or at the beginning of the next) zone. Figures 6 presents the interaction diagram corresponding to the use case *UC1. Take single picture*.

As one may have noticed, in the previous figure, the *Application* represents the external user of the system. In real-life, it is the *Application* that represents the interface between the human user and the system.

Although the messages used in the interaction are asynchronous in nature, some of the messages will have associated a *return message*, which depicts an output message sent by a service in order to confirm or return the status of a

service functionality invoked by a previously received message. Typically, the initiating messages are denoted using the `MessageName_req` format, whereas return messages are denoted using the `MessageName_cnf` format. Exceptions from this rule are allowed when there are more than one return messages for a given request message. For instance, as result of receiving a request message `Capture_req` (see Figure 6), the `SC::StillCapture` lifeline responds first with a `CaptureReady_ind` message confirming that a photo has been taken and transferred to the `Enc::JPEGEncoder`, and then with a *Capture_cnf* message to announce that the JPEG encoded photo has been stored on the *FileServer*.

As shown earlier, energy and timing requirements may be associated with each use case. When the use case spans several subsystems, these requirements must be decomposed to formulate verifiable deadlines and energy budgets for the use case portions executed by the individual subsystems. Such a decomposition is shown in Figure 6. To explore the possibilities to verify these decomposed constraints in practice, we have set up an experimental energy consumption test bench.

The data transfers between services are modeled as input and output streams over the DOA Interconnect, accompanied by parameters describing the transferred data.

4 Tool Support

As previously mentioned, in our subsystem specification approach we employ both a graphical and a textual specification format. The definitions of the two formats are independent, yet they are equivalent. The graphical representation format has been defined using a metamodel and implemented in the Coral modeling tool [4], whereas the XML-based representation has been defined using an XML Schema.

Figure 7 shows a caption of Coral editing the FSM of the *StillCapture* service. We find that especially the protocol state machines and the interaction sequences modeling use cases are far easier to construct and understand using this graphical view of the specifications.

The subsystem, service interface and use case specifications are primarily exchanged between the system designer and vendors as XML files. Although most aspects of these specifications may be successfully edited and validated using any schema-conscious XML editor, we employ Coral to automatically generate the XML-based specification from the existing graphical representations. For instance, the following XML code specifies the *Imaging* subsystem shown in Figure 2.

```
<?xml version="1.0" ?> <Subsystem xmlns="http://mde.abo.fi/NoTASpec/0.1/System/Subsystem"
name="Imaging">
  <documentation>
    This is the imaging subsystem, containing services related to image capture, encoding,
    editing etc.
  </documentation>
```

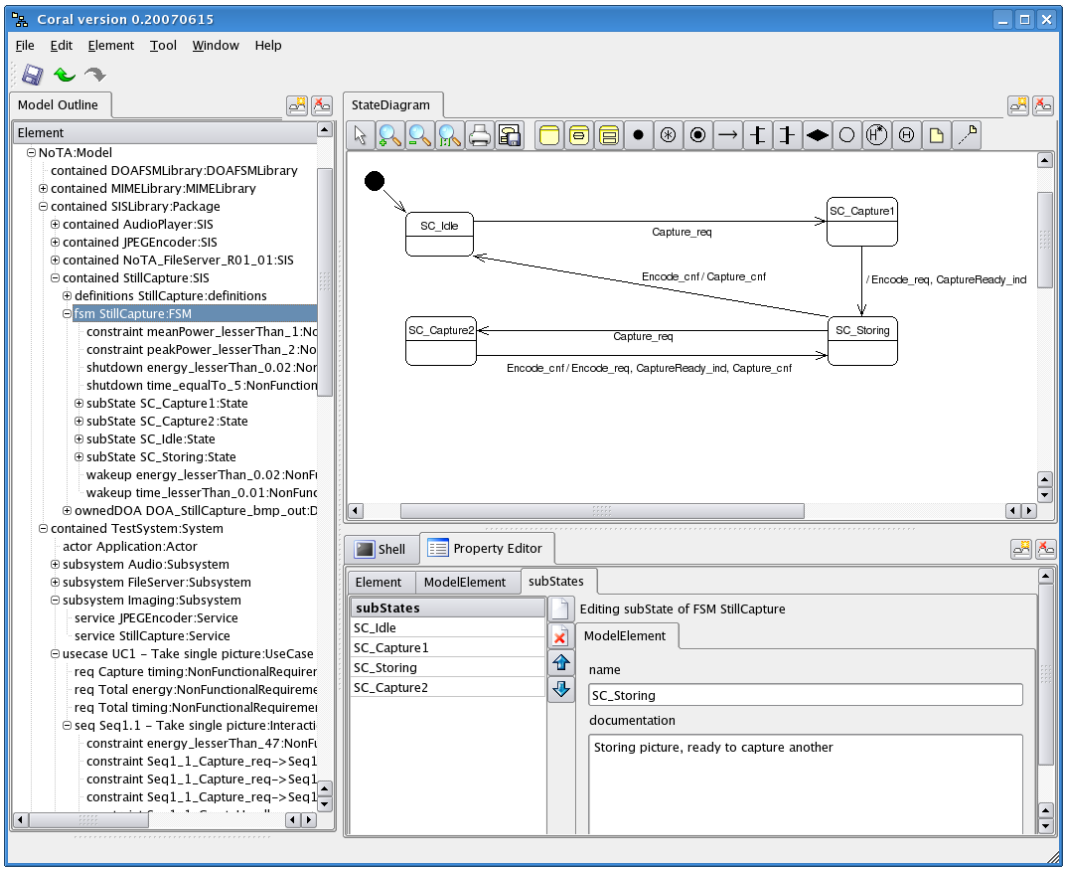

Figure 7: *StillCapture* FSM in Coral

```
  <usecase>UC1 - Take single picture</usecase>
  <usecase>UC2 - Take picture series</usecase>
  <service name="StillCapture" specification="StillCapture">
    <documentation>
      Captures pictures using a camera and encodes them as bmp, no compression, 1280*960.
      May use encoder services to supply other file formats and image sizes.
    </documentation>
    <sequence>Seq1.1 - Take single picture</sequence>
    <sequence>Seq2.1 - Take picture series</sequence>
  </service>
  <service name="JPEGEncoder" specification="JPEGEncoder">
    <documentation>Encodes JPEG images from raw or bmp input. Compression and image width
    should be given, w/h ratio will be preserved.</documentation>
    <sequence>Seq1.1 - Take single picture</sequence>
  </service>
</Subsystem>
```

The XML-code for specifying the *StillCapture* service implemented by the *Imaging* subsystem is shown below. As one may notice, the last part of the code specifies the *StillCapture* FSM corresponding to the one in Figure 4.

```
<?xml version="1.0" ?> <SIS xmlns="http://mde.abo.fi/NoTASpec/0.1" name="StillCapture">
  <documentation>
    Captures pictures using a camera and encodes them as bmp, no compression, 1280*960.
    May use encoder services to supply other file formats and image sizes.
  </documentation>
```

```
<implementation>StillCapture</implementation>
<out>image/bmp</out>
<usesService>JPEGEncoder</usesService>
<ownedDOA name="DOA_StillCapture_bmp_out">
  <dataType>image/bmp</dataType>
  <active Power="0.5" bandwidth="600000000.0" bit_energy="0.0" timeout="0.0016"/>
  <idle Power="0.1"/>
  <shutdown delay="0.001" energy="0.003"/>
  <wakeup delay="0.001" energy="0.003"/>
</ownedDOA>
<definitions name="StillCapture">
  <documentation>
    Captures pictures using a camera and encodes them as bmp, no compression, 1280*960.
    May use encoder services to supply other file formats and image sizes.
  </documentation>
  <message code="0x0001" direction="in" name="Capture_req">
    <documentation>
      Capture a picture and store it on a file server.
    </documentation>
    <part name="width" type="nota:int"/>
    <part name="compression" type="nota:int"/>
    <part name="handle" type="nota:handle_t"/>
  </message>
  <message code="0x1001" direction="out" name="CaptureReady_ind">
    <documentation>
      Indication that a picture has been captured and the service is ready to receive
      another capture request.
    </documentation>
    <part name="status" type="nota:int"/>
  </message>
  <message code="0x1002" direction="out" name="Capture_cnf">
    <documentation>
      The capture is complete, the picture has been encoded and stored on the file server.
    </documentation>
    <part name="status" type="nota:int"/>
    <part name="handle" type="nota:handle_t"/>
  </message>
</definitions>
<fsm initState="SC_Idle" name="StillCapture">
  <documentation>
      FSM for the StillCapture SIS. Some constraints that are not really motivated by the use cases
      were added here in order to illustrate allowed possibilities to define FSM constraints.
  </documentation>
  <wakeup kind="lesserThan" quantity="time" value="0.01"/>
  <wakeup kind="lesserThan" quantity="energy" value="0.02"/>
  <shutdown kind="lesserThan" quantity="energy" value="0.02"/>
  <shutdown kind="equalTo" quantity="time" value="5"/>
  <constraint kind="lesserThan" quantity="meanPower" value="1"/>
  <constraint kind="lesserThan" quantity="peakPower" value="2"/>
  <subState name="SC_Idle">
    <transition name="SC_Capture1_req_transition" nextState="SC_Capture1" trigger="Capture_req">
      <testedBy>Seq1_1_Capture_req</testedBy>
    </transition>
  </subState>
  <subState name="SC_Capture1">
    <documentation>Capturing picture</documentation>
    <constraint quantity="time" value="0.09"/>
    <transition name="SC_Capture1_ready_transition" nextState="SC_Storing">
      <effect>CaptureReady_ind</effect>
      <effect>Encode_req</effect>
      <testedBy>Seq1_1_CaptureReady_ind</testedBy>
      <testedBy>Seq1_1_Encode_req</testedBy>
    </transition>
  </subState>
  <subState name="SC_Storing">
```

```
      <documentation>Storing picture, ready to capture another</documentation>
      <constraint quantity="time" value="0.2"/>
      <transition name="SC_Capture2_req_transition" nextState="SC_Capture2" trigger="Capture_req"/>
      <transition name="SC_Encode1_ready_transition" nextState="SC_Idle" trigger="Encode_cnf">
        <effect>Capture_cnf</effect>
        <testedBy>Seq1_1_Capture_cnf</testedBy>
        <testedBy>Seq1_1_Encode_cnf</testedBy>
      </transition>
    </subState>
    <subState name="SC_Capture2">
      <documentation>Capturing one picture while storing another</documentation>
      <transition name="SC_Capture2_ready_transition" nextState="SC_Storing" trigger="Encode_cnf">
        <effect>CaptureReady_ind</effect>
        <effect>Encode_req</effect>
        <effect>Capture_cnf</effect>
      </transition>
    </subState>
  </fsm>
</SIS>
```

5 Conclusions

We have presented a specification approach for NoTA-based embedded systems aimed at enhancing the understandability of NoTA subsystem specifications by vendors. This reduces not only the communication effort required for the vendor to understand the requirements, but also the risk for the system designer to receive from the vendors subsystem implementations that are not conforming to the specifications.

These specifications are accompanied by testing specifications at different levels of abstraction. At service-level, the FSM of a given service can be used to generate test vectors for verifying the timing and energy consumption of a given service invocation, whereas at subsystem-level, tests are derived from system-level scenarios that are using a given subsystem. These two types of testing specifications are to be employed by vendors to validate that the subsystem implementations are in line with the subsystem specifications. The approach enables the vendor to provide already tested subsystems, thus reducing considerably the testing of the implementations by the system designer. In addition, scenario-based tests are supported in order to enable the testing of the implemented subsystem in integration in the final system.

In order to support the specification process, tool support has been developed to allow the graphical editing of system and subsystem specifications by the designer. In addition, one can generate textual specifications for the subsystems from the system specifications. The generated textual specifications can later on be distributed to selected vendors that will implement the devised subsystems.

It is worth mentioning that some of the artifacts produced throughout the subsystem specification process may be reused in order to speed the development of later NoTA systems. These artifacts should be stored in organized libraries, easily discoverable to designers of new systems. The presented approach promotes

reuse on several abstraction levels. Having in place a collection of specifications the designer can easily create new subsystem specifications reusing existing SISs, or create new systems reusing existing subsystem specifications/implementations. Furthermore, one may reuse existing system-level scenarios in new designs, effectively reusing a set of services and scenario-related requirements.

References

[1] K. Kronlöf, S. Kontinen, I. Oliver, and T. Eriksson. *A Method for Mobile Terminal Platform Architecture Development*. in Advances in Design and Specification Languages for Embedded Systems, pp. 285 – 300, DOI - 10.1007/978-1-4020-6149-3_17. Springer Netherlands, 2007

[2] R. Suoranta, *New Directions in Mobile Device Architectures*. Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06). 2006.

[3] T. Erl, *Service-Oriented Architecture – Concepts, Technology, and Design*, Prentice Hall, 2005.

[4] *Coral modeling tool*.
Online at http://mde.abo.fi.

[5] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.* Network Working Group, September 1996.

[6] W3C, *Web Service Description Language (WSDL)*.
Online at http://www.w3.org/TR/wsdl.

[7] W3C, *XML Schema*,
Online at http://www.w3.org/XML/Schema.

[8] W3C, *WSDL Schema*,
Online at http://www.w3.org/TR/wsdl,.

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-1950-4
ISSN 1239-1891