Device Utilization Analysis for IEC 61499 Systems
in Early Stages of Development

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

Kristian Sandström
Industrial Software Systems
ABB Corporate Research
Västerås, Sweden
kristian.sandstrom@se.abb.com

Abstract

Model-driven and component-based approaches, such as the IEC 61499 standard, allow us to apply analysis to systems in early stages of development. When applied to embedded systems, early analysis can help guide the development process of both the software and the hardware platform, and thus reduce the time and cost of the development. In this paper we present a method for early analysis of device utilization for IEC 61499 systems. The method is based on determining device-specific worst-case execution time of each activity in the application. For this, we use timing information for individual algorithms together with IEC 61499 software and platform models. We provide a prototype implementation integrated in an open-source development environment.

1 Introduction

Many embedded software systems are safety-critical and have a requirement to operate in real-time. For such systems the ability to provide functionality on time is as vital as the functionality itself. One possibility to ensure their correct behavior is through testing and analysis, but these activities are typically not performed until late in system development. The ability to perform early analysis can reduce the risk of redesign in late stages of development, thereby reducing cost and development time.

One approach that allows for early investigation of system properties is analysis on the model level. As this method relies on determining system properties using abstract models, a system does not have to be deployed or even fully implemented to apply the analysis. It also allows for efficient analysis methods, as the analysis is performed on an abstract view of a system rather than on the detailed implementation.

Model-level analysis is highly suitable for model-driven and component-based development methods, in which systems are built using various models. One of the standards that support component-based development for industrial systems is IEC 61499 [5]. The IEC 61499 standard is a proposed successor to IEC 61131-3 [4], one of the most widespread standards used to develop automation software solutions.

This paper presents an approach that uses IEC 61499 models and timing information about individual algorithms to derive utilization information for each device in a system. The utilization analysis is based on compositional worst-case execution time (WCET) analysis of IEC 61499 software models. The compositional model-level approach The analysis can warn developers about possible overutilization of processing units, guide them during system reconfiguration and help dimension the hardware platform. We have tested the analysis method using a prototype tool built as a plug-in to the 4DIAC [11] integrated development environment.

The WCET part of the analysis presented in this paper is an extended version of our previous work, described in detail in [8]. Compared to that work, this paper extends the WCET analysis to consider device types and allocation information, and also addresses the steps from WCET analysis to utilization analysis of the whole application.

The rest of the paper is organized as follows: Section 2 gives an introduction to the IEC 61499 standard. In Section 3 we provide an overview of the our analysis approach. Section 4 describes the data structures that we use during the analysis, while in Section 5 we give a detailed description of the analysis. A description of the prototype implementation is given Section 6. In Section 7 we discuss related work and Section 8 concludes the paper.

2 Background – the IEC 61499 standard

The IEC 61499 standard [5, 12, 15] is proposed as a successor of the IEC 61131-3 standard widely used in industry to accommodate development of industrial automation systems. Over time, the evolvement of controller and network performance have enabled more complex software applications, distributed over multiple controllers, geographically separate in a plant. To meet this increased complexity, the IEC 61499 standard was developed. The
new standard also addresses other high level requirements of new automation systems, such as portability, configurability, interoperability, reconfiguration, on top of distribution of both devices and system intelligence.

2.1 IEC 61499 software model

The main element of the IEC 61499 software model is the function block. Function blocks are reusable units of software that implement a specific functionality with a clear separation between interface and implementation. Considering implementation, a function block can be of three possible types: Basic function block, Service interface function block and Composite function block.

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

Example 1 In Figure 1 a) we can see an example of a function block interface containing event input ports $e_{i11}$ and $e_{i12}$, event output $e_{o11}$, data input port $i_{n11}$ and data output port $o_{u11}$.

A basic function block (BFB) is implemented by means of an Execution Control Chart (ECC) and one or more algorithms. The ECC is an automaton consisting of states and guarded transitions. Each state can be associated with zero or more actions. An action can specify one or more algorithms which will be executed once the state is reached, and output event ports that will be activated. We differentiate between two types of states: stable states in which execution of ECC stops until a new event arrives at an input port, and transitional states which do not require an event for ECC to move to another state. Execution of basic function blocks is strictly event driven – it can only start when an event is received at one of the input ports, and once the execution stops it will not continue until the next event arrives. One execution cycle of a function block is called a run. A single run can traverse more than one ECC state in case the ECC contains transitional states, and thus result in an arbitrary number of algorithm executions and output events.

Example 2 Figure 1 b) shows an example of a basic function block ECC. It contains states START, S1 and S2, with START being the initial state. The transition from START to S1 is guarded by the input event $e_{i11}$. When this state is reached, algorithm A1 is executed, and an event is generated at output event port $e_{o11}$. The transition to state S2 is guarded by the input event $e_{i12}$, and it executes algorithm A2 and outputs an event at $e_{o12}$. Both S1 and S2 are transitional states – there are transitions back to START that are not guarded by any event, depicted by a 1 as the transition condition.

Service interface function blocks (SIFB) are designed to be used as interfaces to external hardware or services. The functionality of this element is not specified by the standard, and although they can contain a sequence diagram describing their behavior, the functionality might not be fully documented. Unlike basic function blocks, the service interface function blocks can start their execution without the arrival of an input event (active execution).

A composite function block (CFB) has an implementation defined by a function block network (defined below), with additional connections between the ports of the enclosing interface and the ports of the function blocks in the network. As the composite function block can contain active service interface function blocks, composites can also be active, i.e. start their execution without receiving an input event.

A function block network (FBN) defines the internal structure of a composite function block or a whole application. A function block network consists of a set of function blocks of arbitrary types (BFB, SIFB or CFB) and connections between the ports of these function blocks. As a result of the separation of event and data ports, the flow of control and data are clearly distinguished.

Example 3 An example of a composite function block can be seen in Figure 1 c). Its internal function block network contains two function blocks, $f_{b1}$ and $f_{b2}$.

An application is the highest hierarchical level of the IEC 61499 software model. It consists of a function block

Figure 1. a) An IEC 61499 function block interface. b) A basic function block ECC. c) A composite function block with the internal function block network.
network and does not have any inputs or outputs at the interface level.

2.2 Processing units in IEC 61499

In IEC 61499 the platform is represented by devices. A device is an independent physical entity capable of performing one or more specified functions. Each device can contain zero or more resources, which are functional units with independent control of operation. If a device does not contain any resources it is considered to be equivalent to a single resource.

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

The specification of the IEC 61499 standard does not clearly define which modeling elements (devices or resources) should represent processing units. One possibility is that processing units are represented by devices. In this case the single processing unit is shared by all resources contained by the device. However, we can also envision that each resource contained by a device corresponds to a separate processing unit (e.g., a CPU, a CPU core or a virtual processing unit). For the purpose of this paper we will assume devices as the processing units. This assumption does not limit the general applicability of our approach, since the same principles can be used to perform utilization analysis for individual resources.

3 Overview of the analysis

The work presented in this paper facilitates the development of systems by providing an efficient method to determine processing resource utilization early in the development process.

The ability for early analysis is addressed by the compositional model-level approach. It not only enables analysis to be performed before a system is deployed to the platform, but also allows analysis of systems that are not fully implemented. The unimplemented parts of the system can be represented by abstract model elements, which can provide estimates of timing properties. During the development, these elements can gradually be replaced by the actual implementation, providing more accurate analysis results.

The efficiency is the result of performing the analysis using abstract models of systems. It is also aided by the compositional approach, which allows reuse of analysis results, eliminating the need to redo analysis of the whole system if only a part of it changes.

We envision the analysis to be integrated in the development environment and automated. In this case it could even be performed continuously, after each change to the system model.

An overview of the method can be seen in Figure 2. It consists of four analysis mechanisms, targeting different levels in the IEC 61499 software model hierarchy. At the bottom level there is an WCET analysis of basic function blocks. The second analysis method targets composite function blocks, using the function block network model and the results of basic or composite function block analysis one hierarchical level below. On the top level of the model hierarchy we perform WCET analysis of applications by using results of function block analysis, while taking into account how the function blocks are mapped over the devices of the platform. Finally, using the results of platform-specific application analysis and information about periods of the execution triggers, we calculate the utilization for each device. Each of the four analysis mechanisms is described in more details in Section 5.

The WCET analysis for each hierarchical level is performed by composing context-independent WCET data of its subcomponents. In case of basic function blocks this is done by analysis of the ECC implementing the block using the WCET of individual algorithms. For composite function blocks and applications the WCET data is obtained by composing the WCET data of the constituent function blocks, according to the event connections in the internal function block network.

Although the analysis of basic and composite function blocks does not use mapping and platform models, we still use separate WCET data for each device type. This allows us to provide appropriate data when we use these models on the application level.

At the bottom of the IEC 61499 model hierarchy are algorithms implemented by code rather than further defined by models. Determining the WCET values for algorithms is outside the scope of this paper, and we assume that this information has been established, either as expert estimates or from code analysis (some of the methods and analysis tools are described in [13]). Similarly, service interface function blocks are also not further elaborated at model level and thus not handled by the analysis. We assume the WCET data for these blocks have been defined manually, e.g., using code analysis, including information about activities originating within the block.

The utilization analysis requires that all system activi-
ties must have a period of their execution. For some execution triggers the periods are predefined, and can therefore be included in the WCET information for the function blocks containing such triggers. However, for many execution triggers the periods can depend on the hardware platform, or the execution can be completely sporadic. In these cases we must manually provide the periods, or use minimum interarrival times of sporadic events as the periods.

4 Representing WCET data

Before giving a detailed description of the analysis mechanisms in Section 5, we will first describe the WCET data for algorithms, function blocks and applications.

4.1 Algorithm WCET data

The analysis assumes that all algorithms used in the implementation of basic function blocks have been assigned WCET values, represented by integer numbers. Because an algorithm can take different amount of time to execute on different devices, we define more than one WCET value for each algorithm, each valid for a specific device type.

Example 4 The WCET of an algorithm A for devices of types \( t_1 \) and \( t_2 \), respectively, can be defined as:

\[
WCET(A, t_1) = 10 \quad WCET(A, t_2) = 12
\]

4.2 Function block WCET data

Compared to algorithms, the WCET data representation for a function block is more complex. It consists of two main data sets: event data and period data. The event data set contains information about execution initiated by the arrival of input events. Therefore, each data entry in this set is associated with an input event port of the function block. The period data set describes execution initiated by the function block’s internal execution triggers, and in this set each data entry is associated with an identifier of the execution trigger.

As the compositional approach to analysis requires the function block WCET data to be context-independent, besides knowing the WCET value for execution of a function block, we must also know what effect the execution can have on the rest of the system. Therefore, each WCET data entry also contains information about generated output events.

In many cases, a particular event (at an input event port or internally triggered) can result in several alternative execution paths, for example depending on the current state of the function block. To cope with cases when it cannot be determined which alternative is in fact the worst case without considering the rest of the system (e.g., one alternative has higher WCET but the other generates more output events), each event or internal trigger can have more than one WCET data entry associated with it, representing the different alternatives.

Example 5 We can illustrate the WCET data for a function block \( fb \) on a device of type \( t \) by the following example:

\[
WCET(fb, t) = \langle \langle e_{i1}, \{\langle 10, \{e_{o1}=1\}\rangle, \langle 5, \{e_{o1}=2, e_{o2}=1\}\rangle \rangle, \langle e_{i2}, \{\langle 30, \{e_{o2}=1\}\rangle \rangle, \langle p_1, \{\langle 3, \{e_{o3}=1\}\rangle \rangle \rangle
\]

The graphical representation of this data can be seen in Figure 3. Each dashed arrow represents a WCET data entry, starting from the input event or an identifier specifying an internal trigger, and pointing to the generated outputs. Multiple outputs at a single event port are represented by multiple arrow heads. The number next to an arrow is the WCET value of that entry.

For input event \( e_{i1} \) we have two WCET data entries. One has a value of 10 and generates one event at the \( e_{o1} \) output port, and the other has a value of 5 but generates two events at \( e_{o1} \) and one at the output port \( e_{o2} \). Input event \( e_{i2} \) has only one data entry with a value of 30 and one event generated at the port \( e_{o2} \). The WCET data also contains one periodical triggering \( p_1 \) which has a single WCET data entry with the value of 3 and one output at \( e_{o3} \).

4.3 Application WCET data

The fact that applications do not have input and output ports (and they cannot be reused in different contexts) makes the WCET data for application much simpler than the one for function blocks. The lack of input ports means that the application analysis only results in WCET information for execution started by internal triggers. Moreover, because there are no outputs there is no need to keep track of alternative execution paths in the result. Instead, we can characterize each internal execution trigger with only one WCET value. Therefore, the WCET data that we will use as the result of application analysis is a set of tuples associating each device with the WCET value that each internal trigger causes on that device.

Example 6 The following exemplifies the representation of WCET data for application app with devices \( d_1 \) and \( d_2 \):

\[
WCET(app) = \{\langle d_1, \{\langle p_1, 20\rangle, \langle p_2, 10\rangle\rangle\}, \langle d_2, \{\langle p_2, 50\rangle\}\rangle\}
\]

The example application has two internal execution triggers. The first trigger \( p_1 \) only results in execution on
with a WCET value of 20. The second trigger \( p_2 \) can cause execution on both devices, for \( d_1 \) the WCET is 10 and for \( d_2 \) it is 50.

5 Processor utilization analysis

This section gives a more detailed description for each of the four analysis mechanisms outlined in Section 3. In the descriptions we will rely on the data structures described in the previous section. Each analysis mechanism will also be demonstrated by an example.

5.1 Basic function block WCET analysis

Basic function blocks are not allowed to start their execution using internal triggers, and therefore we only need to determine the WCET data for execution started by arrival of input events. We do this by analysis of the ECCs that implement their functionality. For each input event port in the interface of a basic function block, we find all possible ECC runs that start with the given event. For each run we sum up the WCET values of executed algorithms and gather information about produced output events. The analysis is always performed with respect to a specific device type, and thus uses the algorithm WCET values which are valid for that type.

While determining ECC runs we only take into account how events guard ECC transitions, while disregarding conditions that depend on input or internal data values. As a result, we differentiate only between transitions that are guarded by a single event and unguarded transitions. We assume that the ECC does not contain cycles consisting of only transitions that are not guarded by an event.

Example 7 The ECC analysis can be illustrated by the example shown in Figure 4.

Here we can see that the single input event of the function block, \( e_{i1} \), can result in two alternative runs. The first run visits only state \( S1 \), while the second one visits states \( S2 \) and \( S3 \).

The analysis of \( fb \), when deployed to a device of type \( t \), would thus result in the following data:

\[
\text{WCET}(fb, t) = \{ \{ e_{i1}, \{ 10, \{ e_{o1} = 1 \} \}, \{ 8, \{ e_{o1} = 1, e_{o2} = 1 \} \} \} \}, \emptyset
\]

5.2 Composite function block WCET analysis

Execution of a composite function block can either start by the arrival of an input event or by an internal execution trigger in one of the function blocks in its function block network (e.g. a service interface function block). The WCET analysis for these two types of execution is done separately. However, both parts rely on the same analysis of the function block network.

The analysis of executions based on the arrival of input events starts from the interface of the composite function block. Similarly to the analysis of basic function blocks, for each input event port we find all possible execution paths in the function block network of the composite. The execution paths are determined by traversing event connections and by using the event WCET data entries (more precisely, the event output information of the entries) of the function blocks contained by the network. For each execution path, the WCET values of the utilized event WCET data entries are accumulated. If during the analysis we reach an output event port of the composite, we increase the occurrences of that event in the resulting information about generated outputs.

The first step of the analysis for execution initiated by internal execution triggers is finding all function blocks in the network that have at least one entry in their period data sets of their WCET. For each such entry we perform the same network analysis as described earlier, but using the event output information of the entry as starting points.

Performing analysis for a specific device type requires that for each function block inside the composite we have WCET data valid for that type. If such data is not present for a function block, we must first acquire it by performing device-specific analysis on that block.

Example 8 We describe how composite function block analysis is performed on the example in Figure 5.

The event WCET data for the composite \( cfb \) is determined by starting the analysis at the input event port \( e_{i1} \). From this event port we can trace two different execution paths. Although both paths visit same function block on this hierarchical level, they take different internal paths (WCET data entries) in \( fb_2 \).
The single entry of the period WCET data set for the composite is calculated from the period data entry for \( f_{b3} \), by following the execution path to the outer port \( e_{oc3} \).

The resulting WCET data for \( cf_{b} \) and device type \( t \) is:

\[
\text{WCET}(cf_{b}, t) = \{ \{ e_{ic1}, \{ \{ 111, \{ e_{oc1}=2, e_{oc2}=1 \} \}, \{ 131, \{ e_{oc1}=1, e_{oc2}=1 \} \} \} \}
\]

\[
\{ \{ p_{12}, \{ 5, \{ e_{oc3}=1 \} \} \}\}
\]

5.3 Application WCET analysis

As IEC 61499 applications do not have input or output event ports in their interfaces, we only have to analyze the WCET of execution based on the internal triggers of the function blocks that implement the application. This analysis is performed using the same principles as for the composite function blocks, described in the previous section.

However, when performing analysis on the application level we must also take into consideration the distribution of function blocks to devices. To achieve this, the analysis is performed once for each device in the platform. In each run we ignore the WCET values of function blocks which are not mapped to that device, and for function blocks which are mapped to that device we use the WCET data which is valid for that device type.

While during the analysis for a specific device we use only WCET values for function blocks mapped to this device, we still use the output information and periods of internal execution triggers for all function blocks. In this way we still take into account all possible execution paths, but return as the result the ones that produce maximal WCET values for the analyzed device rather than the ones that result in the maximal overall WCET in the whole system.

Example 9 We will show an example of application analysis using the application depicted in Figure 6 a). The example platform consists of two devices, \( d_{1} \) and \( d_{2} \), and for the purpose of simplicity, we will assume that both devices are of the same type. We further assume that function blocks are mapped to devices as specified by Mapping A in Figure 6 b).

If we perform WCET analysis for the example application, we get the following results:

\[
\text{WCET(app)} = \{ \{ d_{1}, \{ \{ p_{11}, 115 \}, \{ p_{12}, 5 \} \} \}, \{ d_{2}, \{ \{ p_{11}, 60 \} \} \}
\]

5.4 Calculation of processor utilization

Once WCET values for applications with regards to individual devices have been determined, and periods for each execution origin are available, we can proceed to calculate utilization for the devices.

Utilization is calculated for each device separately. For each execution trigger we calculate its utilization contribution by dividing the WCET value by the period of the trigger. The result of the utilization analysis for the device is the sum of utilization contributions of all execution triggers. The utilization of device \( d \) in application \( app \) can thus be defined as:

\[
U(app, d) = \sum_{(p,w) \in S} \frac{w}{\text{period}(p)},
\]

where \((d, S) \in \text{WCET(app)}\).

The result of utilization analysis is a positive real number, with the value of 1 representing full utilization of a device. A value over 1 corresponds to overutilization.

Example 10 To demonstrate the utilization analysis we will again use the application from Figure 6 a), assuming that the platform consists of two devices \( d_{1} \) and \( d_{2} \), of the same type. To show how we can detect overutilization of a device, and verify that the problem is solved after redistributing function blocks between the two devices, we will use two different function block mappings. The details of the two mapping alternatives are given in Figure 6 b). As period values for \( p_{11} \) and \( p_{12} \) we will use 300 and 500, respectively. Table 1 shows the utilization analysis results.

From the results we can see that when using mapping A the device \( d_{2} \) is overutilized, having the utilization value 1.2. Changing the mapping of the function blocks to option B resolves the problem of overutilization, as both devices have the utilization value lower than 1.

It should be noted that the sum of WCET for all devices and execution triggers is not the same in mapping A and B. This is because the analysis considers the worst case condition.
for each device in isolation. For mapping A, only one of the two alternative paths in fb₂ contributes to the worst case. For mapping B, however, both of them result in worst cases, for device d₁ and d₂ respectively.

Table 1. Results of the utilization analysis.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>d₁</td>
<td>p₁₁ (300)</td>
<td>115</td>
<td>0.383</td>
<td>0.483</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁₂ (50)</td>
<td>5</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d₂</td>
<td>p₁₁ (300)</td>
<td>0</td>
<td>0</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁₂ (30)</td>
<td>60</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>d₁</td>
<td>p₁₁ (300)</td>
<td>85</td>
<td>0.283</td>
<td>0.983</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁₂ (50)</td>
<td>35</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d₂</td>
<td>p₁₁ (300)</td>
<td>100</td>
<td>0.333</td>
<td>0.933</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁₂ (50)</td>
<td>30</td>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>

6 Implementation

The presented processing unit utilization analysis is supported by a prototype tool. An earlier version of the prototype, supporting only WCET analysis without the ability for resource- or device-specific execution, was presented in [8].

The analysis tool was developed as a plug-in for the 4DIAC-IDE [11], an open-source graphical development environment for IEC 61499. The prototype allows executing utilization analysis for both resources and devices using the graphical editors of the IDE. The implementation currently does not support storing device-type-specific WCET data for each function block, but uses the same data for all device types.

An earlier prototype implementation was used to test the applicability of the WCET analysis method by a set of experiments (described in details in [8]). In the experiments we applied non-device-specific application analysis to four systems from the 4DIAC and FBDK sample libraries. The results showed that such analysis can be performed in the order of magnitude of 10 milliseconds even for non-trivial systems. The new implementation shows similar performance when executing the device utilization analysis. This indicates that the method is fast enough to be automatically executed by the development environment, as a reaction to every change of the system model.

7 Related work

An introduction to WCET analysis can be found in the work by Wilhelm et al. [13]. Lisper [9] describes the potential and problems in model-level WCET analysis, as well as the benefits of early, approximate analysis.

The need for early analysis is also stressed in work by Gustafsson et al. [3] where the authors propose a method for early WCET analysis on code level by first creating a timing model of the code. Compared to the work presented in this paper, this method does not allow performing the analysis on a system level (taking into consideration the software and hardware platform at the same time), and does not produce safe estimates.

An approach for WCET analysis on an architectural level using AADL is presented by Gilles and Hugues [2]. The method first performs a model-to-code transformation, and then performs the analysis on the code level. The results of the analysis are in the end propagated back to the model.

Parts of the analysis presented in this paper were inspired by the approach for WCET analysis for the ProCom component model, described in [1]. Compared to the WCET part of the analysis presented in this paper, the analysis for ProCom does not allow having multiple WCET data entries for a single event port and does not utilize platform and mapping information during the analysis.

A method for worst-case reaction time analysis in IEC 61499 systems is described by Kuo et al. [7]. The authors first compile the system to gather timing information for its code. The analysis continues by applying a model-checker on the system code to incrementally predict the reaction time, while visiting all possible states of the system.

Schedulability analysis [10] is used in real-time applications to determine that all timing requirements of a system are satisfied.

Khalgui et al. present a schedulability analysis for IEC 61499 systems [6]. The analysis method relies on transforming systems to OS tasks. Each task is characterized by a WCET value and its predecessors and successors. Arranging the tasks in chains allows one to check if end to end deadline are violated and generate off-line scheduling for systems.

In [14] Zoitl et al. define the concept of event chains as sets of function block execution, starting from an event source at a service interface function block and ending with a function block which will not cause any further execution. The authors aim to allow application of real-time scheduling theories by assigning real-time constraints to these event chains.

8 Conclusion

We have presented a method for analysis of device utilization in IEC 61499. The analysis is performed on IEC 61499 system models. Using device-type-specific timing information for the algorithms we calculate WCET data for basic and composite function blocks. Using information about mapping of applications to devices, for each platform device we calculate device-specific WCET values for all internal execution triggers of the applications allocated to run on the device. These values, together with the periods of the execution triggers, are used to determine the utilization of a device.

Performing the analysis using system models allows us to calculate device utilization while the systems are not deployed or even fully implemented. The efficiency of the analysis is aided by the compositional approach which
supports the reuse of function block analysis results. Because of this the analysis can be performed often, and during the whole development process.

Future work will include full implementation of the analysis method, which would allow automatic continuous analysis of systems during the development process.

Our plans for future work also include experiments that would compare the utilization approximations of our analysis to approximations provided by static code analysis, or measurements on deployed systems.

9 Acknowledgments

This work has been performed in the ASSIST project at Mälardalen University, funded by the ABB Software Research Grant Program, and the Ralf3 project funded by the Swedish Foundation for Strategic Research.

References