16th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE'13), Vancouver, BC, Canada, June 17–21, 2013. ACM.

Model Level Worst-Case Execution Time Analysis for IEC 61499

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

The IEC 61499 standard provides a possibility to develop industrial embedded systems in a component-based manner. Besides alleviating the efforts of system design, the component-based approach also allows analysis of various system characteristics using system models even before the actual deployment. One of the crucial characteristics in the domain of safety-critical and real-time systems is timing: a failure to execute a specific task on time can have severe consequences.

This paper presents a method for compositional model-level analysis of worst-case execution time of IEC 61499 software models. The analysis is performed on one hierarchical level of composition at a time, and the results can be stored together with the software artefact to be used when analysis is performed on the higher hierarchical level, or when the unit is reused in another system. The analysis has been implemented as a plug-in for the 4DIAC tool.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

IEC 61499, timing, analysis, WCET

1. INTRODUCTION

One of the most widespread standard used in industrial software systems, such as manufacturing, rolling mills and mining hoists, is currently the IEC 61131-3 [4]. However, a lot of effort has been invested into developing its possible successor, the IEC 61499 standard [5]. The new standard introduces more powerful execution model and better system modeling support, allowing creation of software with more complex functionality.

The ability to support more complex systems also requires more advanced analysis methods for understanding their execution behavior. One possible technique that can be applied to the IEC 61499 standard is model-level analysis. It allows for more efficient analysis algorithms as it is performed on an abstract view of a system. It can also be applied on only parts of the system, or in early stages of development, even before the system is fully implemented and deployable. Still, analysis methods determining functional and non-functional properties of IEC 61499 systems are very rare, as currently this is not one of the primary concerns of the standard [13, 15].

An important property that we can analyze in real-time embedded software systems is the Worst Case Execution Time (WCET). The WCET value corresponds to the maximum time that a processing resource will be used to finish execution of a task or a component's functionality [10]. Knowing the WCET for the components of a real-time system is essential for ensuring it's timing requirements are met.

In this paper we will describe a method for model-level WCET analysis for IEC 61499 systems. We start the analysis on the bottom level of hierarchy and perform it in a compositional manner. The WCET data for each model element is calculated by composing the data of its subcomponents, based on the models describing their interaction. Once the analysis of an element is done we attach the resulting context-independent data to that element. We then use these results when an instance of the element is encountered while applying the analysis on higher levels of the hierarchy. To address a possible problem of scalability we provide different versions of the analysis, with different precision and resource usage. A prototype analysis tool has also been implemented on top of the 4DIAC environment for development of IEC 61499 systems.

The rest of the paper is organized as follows: Section 2 gives an overview of the software model for the IEC 61499 standard and a formal definition of the model elements needed to define our analysis algorithms. The analysis method, including formal definitions of data and algorithms, is given in Section 3. In Section 4 we describe our prototype analysis tool and Section 5 presents experimental results. Related work is discussed in Section 6 and finally Section 7 concludes the paper.

2. THE IEC 61499 STANDARD

The IEC 61499 standard 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 has been 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 architectural elements

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 of the system, but do not provide any means for exchanging data between function blocks. All data transfers are done by data inputs and outputs.

EXAMPLE 1. *Figure 1 a) shows an example of a function block interface containing event input ports e_{i11} and e_{i12}, event output e_{o11}, data input port in_{11} and data output out_{11}.*

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 an algorithm which should be executed once the state is reached, and an output event port 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 the input ports, and transitional states which do not require an event for ECC to move to another state. Basic function blocks are strictly event driven – the execution 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.

The analysis method presented in this paper does not take into account values of data ports or internal variables, making them irrelevant from the analysis point of view. For the purpose of the analysis we transform all ECC transition guards by removing the parts of the guard conditions that include data values. This means that after the transformation each transition is either guarded by exactly one event input, or left unguarded. We also assume that the ECC does not contain any event-free cycles, i.e. cycles that do not contain at least one transition guarded by an input event. As the analysis does not take into account data values, we cannot analyze ECCs containing such loops.

EXAMPLE 2. *In Figure 1 b) we can see an example of a basic function block ECC. It contains states START, S1 and S2, with state START being the initial state. The transition from START to S1 is guarded by the input event e_{i1}. When this state is reached, algorithm A1 is executed, and an event is generated at output event port e_{o1}. The transition to state S2 is guarded by the input event e_{i2}, and it executes algorithm A2 and outputs an event at e_{o2}. Both S1 and S2 are transitional states – there are transitions back to START that are not guarded by any event, their guard expression being represented by the value 1.*

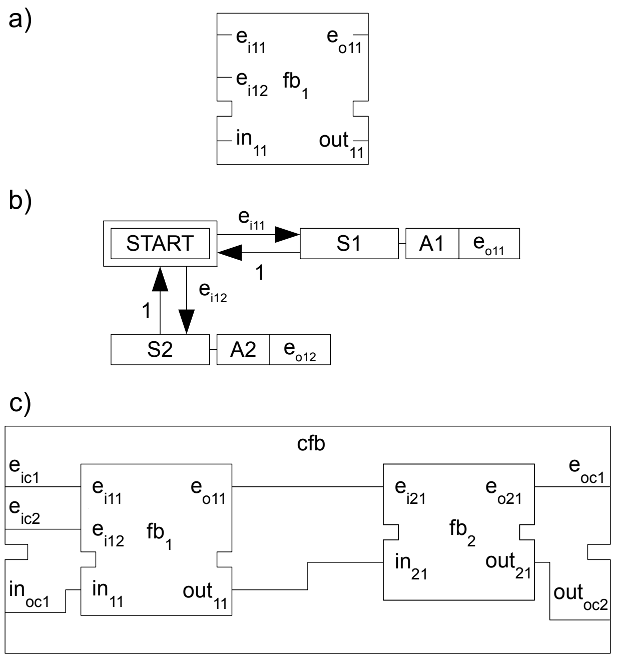


Figure 1: a) A IEC 61499 function block interface. b) Example of a basic function block ECC. c) An example of a composite function block with the internal function block network.

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 as a result of internal execution triggers, without the arrival of an input event (active execution). The occurrence of such internal triggers can either have a fixed period, or it can be sporadic.

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.

In some cases our analysis cannot handle function block networks containing event cycles, depending on the behavior of the function blocks contained by the cycle, for example if the termination of the cycle depends on data values.

EXAMPLE 3. *An example of a composite function block can be seen in Figure 1 c). It's internal function block network contains two function blocks, fb_1 and fb_2.*

For the purpose of this work we view *applications* as composite function block with empty interfaces.

2.2 Formal Definition of IEC 61499

To be able to define the WCET analysis, we first have to provide a formal definition of the IEC 61499 elements used in the analysis. Parts of the formal definition provided in this paper are based on work from Čengić and Åkesson [12]. Definitions that differ from the ones defined in the mentioned paper were modeled according to the IEC 61499 standard definition [5]. In this definitions we will disregard elements that are not relevant to the analysis described in this paper, such as data ports and connections between them.

Definition 1. The following constructs constitute our formal representation of IEC 61499 systems:

Function block interface: $fbi = \langle E_i, E_o \rangle$ where

E_i	is a set of event inputs;
E_o	is a set of event outputs.

Basic function block: $bfb = \langle fbi, ecc, A \rangle$ where

fbi	is a function block interface;
ecc	is an execution control chart (ECC);
A	is a set of algorithm functions.

Execution control chart (ECC): $ecc = \langle Q, T \rangle$ where

Q	is a set of ECC states;
T	is a set of ECC transitions.

ECC state: $q = \langle k_1, ..., k_{|q|} \rangle$ where

k_i	is a ECC state action;

ECC state action: $k = \langle a, e_o \rangle$ where

a	is the algorithm to be executed, $a \in A$;
e_o	is the output event to be generated, $e_o \in E_o$.

ECC transition: $t = \langle q_s, e_g, q_d \rangle$ where

q_s	is the source state, $q_s \in Q$;
e_g	is the input event guarding the transition, or 1 if the transition is not guarded by an event, $e_g \in E_i \cup \{1\}$;
q_d	is the destination state, $q_d \in Q$.

Service interface function block: $sifb = \langle fbi \rangle$ where

fbi	is a function block interface;

Composite function block: $cfb = \langle fbi, fbn \rangle$ where

fbi	is the function block interface;
fbn	is the internal function block network.

Function block network: $fbn = \langle F, C \rangle$ where

F	is a set of function blocks, each of which can be either a *bfb*, *sifb* or *cfb*;
C	is a set of connections between event ports.

Connection: $c = \langle e_s, e_d \rangle$ where

e_s	is the event port used as the source of the connection;
e_d	is the event port used as the connection target.

3. WCET ANALYSIS

There are two main parts of our analysis method: *basic function block analysis* and *composite function block analysis.* Analysis of basic function blocks is performed first, as they are at the bottom of the function block hierarchy. Their WCET data, containing WCET values and information about generated output events, is derived by analysis of the ECC. The composite function block analysis produces the same data by combining the data of function blocks contained in the function block network, based on the event connections in the network.

As we have already noted, for the purpose of this paper we can view applications as composite function blocks with empty interfaces. Because of that, we do not need to provide specialized algorithms for their analysis, but use the ones defined for composite function blocks instead.

Before describing the actual analysis, we first give definitions for WCET data used in the analysis, together with definitions of operations on that data.

3.1 WCET Data Definition

To enable compositional WCET analysis of IEC 61499 software systems we must first define the format for the reusable, context-independent data that we will use to describe the WCET of a function block.

The WCET data for function blocks is organized in two main data sets: (a) *event WCET data* and (b) *period WCET data.* The event data set consists of entries with information about execution initiated by the arrival of event inputs to the function block. These data entries contain a WCET value and the number of events produced at each event output of the function block. The WCET value represents the maximum amount of time that a function block will require a processing resource to execute its functionality, from the arrival of an input event until the resulting activity of the function block finishes. This time does not include waiting due to preemption or blocking. Because an event input can result in different internal execution paths, and thus in different execution times and output events, there can be more than one WCET data entry associated with each input event. Storing WCET data for different execution path is necessary because when considering the function block in isolation we do not know which alternative leads to the worst overall system WCET.

Similarly, the period WCET data contains information for executions initiated by the internal activities in the function block. In this case the WCET data entry sets are associated with a period of execution instead of an input event. For sporadic internal execution triggers, the minimum interarrival time is used as period value. If the minimum interarrival time is not know, the period value is set to -1.

Once we calculate the WCET data of a function block it can be reused in any context, and needs to be recalculated only if the internals of the function block change.

Definition 2. The WCET data for an algorithm $a \in A$ and a function block f are represented by the functions $\text{WCET}(a)$ and $\text{WCET}(f)$, respectively:

Algorithm WCET *data:* $\text{WCET}(a) \in \mathbb{N}$

Function block WCET *data:* $\text{WCET}(f) = \langle W_e, W_p \rangle$ where

- W_e is a set of elements on the form $\langle e_i, W \rangle$, representing WCET information for input events.
- W_p is a multiset of elements on the form $\langle p, W \rangle$, representing WCET information for periodical execution.
- e_i is an input event, $e_i \in E_i$;
- p is the period of an internally triggered execution, $p \in \mathbb{Z}^+ \cup -1$;
- W is a set of WCET data entries.

WCET *data entry:* $w = \langle v, o \rangle$ where

- v is the WCET value, $v \in \mathbb{N}$;
- o is a function mapping output ports to the maximum number of events generated at them, $o \subseteq E_i \times \mathbb{N}$.

EXAMPLE 4. *We can illustrate the* WCET *data for a function block by the following example:*

$$\begin{aligned}\text{WCET}(fb) = \langle \{ & \langle e_{i1}, \{ \langle 10, \{e_{o1}{=}1\} \rangle, \\ & \qquad \langle \ 5, \{e_{o1}{=}2, e_{o2}{=}1\} \rangle \ \} \rangle, \\ & \langle e_{i2}, \{ \langle 30, \{e_{o2}{=}1\} \rangle \ \} \rangle \ \}, \\ \{ & \langle 50, \{ \langle \ 3, \{e_{o1}{=}1\} \rangle \ \} \rangle \ \} \rangle\end{aligned}$$

In the example, the o functions are shown as sets of equations between ports and number of generated events at that port, and for all output events that do not appear in the set the value of the function is 0. We can see that 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. The second one has a value of 5 but generates two events at e_{o1} and one event 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 output port e_{o2}. The* WCET *data also contains one periodical execution with a period of 50 and a single* WCET *data entry with the value of 3 and one output at the e_{o1} port.*

For the purpose of our analysis we also need to define the following operations on the data elements:

Definition 3. For the functions o, and $n \in \mathbb{N}$, we define the operations $+$, $*$ and inc as follows:

$$\begin{aligned}\text{inc}(o, e', n)\ (e) &= \begin{cases} o(e) + n & \text{if } e = e' \\ o(e) & \text{otherwise} \end{cases} \\ (o_1 + o_2)\ (e) &= o_1(e) + o_2(e) \\ (o * n)(e) &= n * o(e)\end{aligned}$$

Definition 4. For the sets of WCET data entries, W in the definition above, and for $n \in \mathbb{N}$, we define the following operations:

$$\begin{aligned}W * n &= \{ \langle v * n, o * n \rangle \mid \langle v, o \rangle \in W \} \\ W_1 \otimes W_2 &= \begin{cases} W_1 & \text{if } W_2 = \emptyset \\ W_2 & \text{if } W_1 = \emptyset \\ \{ \langle v_1 + v_2, o_1 + o_2 \rangle \mid & \\ \quad \langle v_1, o_1 \rangle \in W_1 \wedge & \text{otherwise} \\ \quad \langle v_2, o_2 \rangle \in W_2 \ \} & \end{cases}\end{aligned}$$

3.2 Data normalization

As we have shown in Section 3.1, WCET data that we define can contain more than one data entry for the same execution source (i.e. input event or internal trigger). When using such data sets in compositional analysis the amount of resulting data can grow rapidly, when all combinations of alternatives for all subcomponents must be considered. To address this problem we will use data normalization to remove redundant data entries and optimize large data sets by introducing over-approximations.

First, we introduce a comparison relation capturing when one WCET data entry is completely covered by another one.

Definition 5. We define the following comparison relation between $w_1 = \langle v_1, o_1 \rangle$ and $w_2 = \langle v_2, o_2 \rangle$:

$$\begin{aligned}w_1 \preceq w_2 &\overset{\text{def}}{=} v_1 \leq v_2 \wedge \forall e : o_1(e) \leq o_2(e) \\ w_1 \prec w_2 &\overset{\text{def}}{=} w_1 \preceq w_2 \wedge w_1 \neq w_2\end{aligned}$$

In our current work we have defined two methods of data normalization, the *maximal elements method* and the *supremum method*, which we will now describe.

3.2.1 The maximal elements method

Using the maximal elements method we only remove redundant data from our WCET data sets. By this method we normalize a data set W by keeping only entries which are maximal elements of W. Since all other elements are guaranteed to produce lower or equal WCET values in any context, and thus can be removed without loss of precision, this method allows reducing the number of data entries without introducing any overestimation in the normalization process.

Definition 6. The maximal elements normalization function is defined as follows:

$$\text{NORMALIZE}^{\max}(W) = \{ w \mid w \in W \wedge \neg \exists w' \in W : w \prec w' \}$$

3.2.2 The Supremum Method

The second normalization method deals with incomparable WCET data values by replacing them by the supremum (the least upper bound), i.e. the smallest value which is greater than both of them. The result of such normalization can be a drastic reduction of data and complexity of analysis, as all alternative execution paths for the input event or periodic activity are represented by a single WCET entry. However, this method also introduces an overestimation which can grow during every normalization, and can decrease the precision of the analysis results.

Definition 7. The supremum normalization function is defined as follows:

$$\text{NORMALIZE}^{\sup}(W) = \{ \sup(W) \}$$

EXAMPLE 5. *As an example of the two normalization methods, consider the following set of* WCET *data entries:*

$$\begin{aligned}W = \{ & \langle 10, \{e_{o1}{=}2\} \rangle, \\ & \langle \ 8, \{e_{o1}{=}1, e_{o2}{=}1\} \rangle, \\ & \langle \ 3, \{e_{o1}{=}2\} \rangle \}\end{aligned}$$

For this set, the two normalization methods give the following results:

$$\begin{aligned}\text{NORMALIZE}^{\max}(W) &= \{ \langle 10, \{e_{o1}{=}2\} \rangle, \\ & \qquad \langle \ 8, \{e_{o1}{=}1, e_{o2}{=}1\} \rangle \} \\ \text{NORMALIZE}^{\sup}(W) &= \{ \langle 10, \{e_{o1}{=}2, e_{o2}{=}1\} \rangle \}\end{aligned}$$

The maximal elements normalization removes the third element since it is smaller than the first element, while the supremum method returns a single element that safely approximates all three.

3.3 Basic Function Block Analysis

Now that we have described the WCET data that we will use in the analysis and the operations for manipulating this data, we can present the actual analysis method. This section will describe the first part of our analysis method, the analysis of basic function blocks. Analysis of composite function blocks will be given in the Section 3.4.

The WCET of a basic function block is determined by analysis of its ECC. To gather the data about execution based on input events, we must analyze all possible ECC runs that can be executed by the event input ports in the function block interface. As the execution of a basic function block can only start by receiving an input event, their WCET data only contains the event data set W_e, while the W_p set of information for periodical execution is always empty.

We start the basic function block analysis by going through the interface, and for each input event port we look for all transitions guarded by the given event. For each such transition we go through all possible ECC runs, and for each run add together the WCET values of the algorithms to be executed and collect information about produced output events.

EXAMPLE 6. *We can illustrate how the ECC analysis is performed on the example shown in Figure 2. We will assume that the ECC in the figure is part of a basic function block bfb_1 containing only one input event port, e_{i1}, and two output event ports, e_{o1} and e_{o2}. Here we can see that the event e_{i1} can result in two different runs. The first run visits only state S1, executes algorithm A1 and produces an event on the output port e_{o1}. The second run visits states S2 and S3, executes algorithms A2 and A3, and produces events at both outputs e_{o1} and e_{o2}.*

As the result of the analysis of bfb_1 we would get the following data:

$$\begin{aligned}\mathrm{WCET}(bfb_1) = \langle\{\ \langle e_{i1}, \{\ &\langle 10, \{e_{o1}=1\}\rangle, \\ &\langle\ 8, \{e_{o1}=1, e_{o2}=1\}\rangle\ \}\rangle\ \}, \\ \emptyset\ \rangle\end{aligned}$$

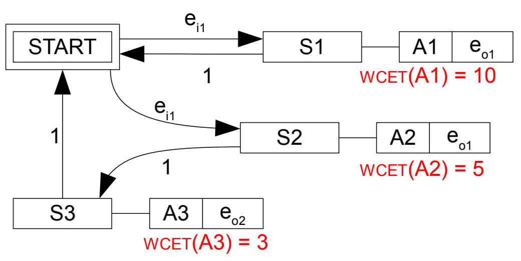


Figure 2: ECC analysis example.

3.3.1 The BFB Analysis Algorithm

Algorithm 1 performs the analysis of a basic function block. The *for* loop starting on line 3 is used to iterate over all event inputs. The combination of *for* loop on line 5 and an *if* statements on line 6 is used to find all transitions that are guarded with the given input event. Then, on line 7, we call the ECC analysis function (described in Algorithm 2) to collect analysis results for all ECC runs starting from the destination state of a transition. The results from the ECC analysis are added to a set W, which will in the end be associated with currently analyzed input event and added to the result set W_e on line 11.

Algorithm 1 BFBANALYSIS(bfb)

```
 1: $W_e \leftarrow \emptyset$
 2: $\langle\langle E_i, E_o\rangle, \langle Q, T\rangle\rangle \leftarrow bfb$
 3: for each $e_i \in E_i$ do
 4:     $W \leftarrow \emptyset$
 5:     for each $\langle q_s, e_g, q_d\rangle \in T$ do
 6:         if $e_g = e_i$ then
 7:             $W \leftarrow W \cup \mathrm{ECCANALYSIS}(q_d,\ T)$
 8:         end if
 9:     end for
10:     $W \leftarrow \mathrm{NORMALIZE}(W)$
11:     $W_e \leftarrow W_e \cup \{\langle e_i, W\rangle\}$
12: end for
13: return $\langle W_e, \emptyset\rangle$
```

3.3.2 The ECC Analysis Algorithm

The ECC run analysis is described by Algorithm 2. In lines 1 to 6 we collect the WCET value v for all algorithms and output information o of actions defined for current ECC state. We use the combination of a *for* loop and an *if* statement on lines 8 and 9 to find all ECC transitions starting with current state that do not have any event inputs as guards. We recursively start ECC analysis for destination states of such transitions. The results of recursive analysis are stored in a temporary WCET data entry set, W'. If there were no such results (i.e. no possible transitions), we assign the WCET value and output information for the current state as the final ECC analysis results on lines 13 and 14. Otherwise, we add the data for the current state to all WCET data

Algorithm 2 ECCANALYSIS(q, T)

```
 1: $v \leftarrow 0$
 2: $o \leftarrow \emptyset$
 3: for each $\langle a, e_o\rangle \in q$ do
 4:     $v \leftarrow v + \mathrm{WCET}(a)$
 5:     $o \leftarrow \mathrm{inc}(o, e_o, 1)$
 6: end for
 7: $W' \leftarrow \emptyset$
 8: for each $\langle q_s, e_g, q_d\rangle \in T$ do
 9:     if $q_s = q$ and $e_g = 1$ then
10:         $W' \leftarrow W' \cup \mathrm{ECCANALYSIS}(q_d,\ T)$
11:     end if
12: end for
13: if $W' = \emptyset$ then
14:     $W \leftarrow \{\langle v, o\rangle\}$
15: else
16:     $W \leftarrow \emptyset$
17:     for each $\langle v', o'\rangle \in W'$ do
18:         $W \leftarrow W \cup \{\langle v + v', o + o'\rangle\}$
19:     end for
20:     $W \leftarrow \mathrm{NORMALIZE}(W)$
21: end if
22: return $W$
```

entries collected by the recursive analysis of ECC in lines 16 to 20. Before we return the result data set W, we normalize its data in line 20.

3.4 Composite Function Block Analysis

Analysis of a composite function block consists of two separate parts: (a) analysis of execution based on input events and (b) analysis of the internal periodic execution sources. Both of these parts are based on the analysis of the function block network contained in the composite function block.

Input event execution analysis starts from the interface of the composite function block. Similar to the analysis of basic function blocks, for each input event we find all possible execution paths in the internal function block network. The analysis is only done on one hierarchical level. We determine the execution paths by traversing event connections of the network and by using event output information included in the existing WCET data for the function blocks in the network. For each execution path we accumulate the WCET values defined in the function block WCET data and gather information about produced output events if a path ends at one or more output event ports of the composite.

Analysis of the internal event sources is performed by iterating over all function blocks contained in the network and finding the ones which have at least one entry in their periodic WCET data set. For each such entry we start a network analysis based on the event output information of the entry.

EXAMPLE 7. *We will illustrate the analysis of composite function blocks by a simple example depicted in Figure 3. The figure shows a composite, cfb, containing three function blocks, and we assume the following* WCET *data for them:*

$$\text{WCET}(fb_1) = \langle \{ \ \langle e_{i11}, \{ \ \langle 1, \{e_{o11}{=}1, e_{o12}{=}2\} \rangle \ \} \rangle \ \}, \ \emptyset \rangle$$

$$\text{WCET}(fb_2) = \langle \{ \ \langle e_{i21}, \{ \ \langle 10, \{e_{o21}{=}2\} \rangle, \ \langle 30, \{e_{o21}{=}1\} \rangle \ \} \rangle \ \}, \ \emptyset \rangle$$

$$\text{WCET}(fb_3) = \langle \{ \ \langle e_{i31}, \{ \ \langle 300, \{e_{o31}{=}1\} \rangle, \ \langle 100, \{e_{o31}{=}1, e_{o32}{=}1\} \rangle \ \} \rangle \ \}, \ \{ \ \langle \ 50, \{ \ \langle \ 10, \{e_{o31}{=}1\} \rangle \ \} \rangle \ \} \rangle$$

This WCET *information is also represented graphically in the figure. Each dashed arrow represents a* WCET *data entry, starting from the input event and pointing to the generated outputs. Multiple outputs at a single event port are represented by multiple arrow heads. The underlined number next to an arrow is the* WCET *value of the entry.*

The WCET *data for the composite cfb is determined by starting the analysis at the input event port e_{ic1}. The first function block in the event path is fb_1. As we can see, the single* WCET *data entry for the port e_{i11} of fb_1 has a value of 1, one output to e_{o11} and two outputs to e_{o12}. The analysis continues by first gathering information for all execution paths starting with e_{o11} and e_{o12}.*

For the first port, the event path leads to fb_2. It has two WCET *data entries, both generating output only on e_{o21}. As this output is connected to the event port e_{oc1} of the containing composite, we stop the analysis at this output. The resulting data entries for the execution starting with e_{o11} are the same as the ones for port e_{i21}, with the outer port e_{oc1} replacing the port e_{o21}, because of the direct event connection from e_{o21} to e_{oc1}. The figure shows these entries above the event connection, together with an arrow describing their*

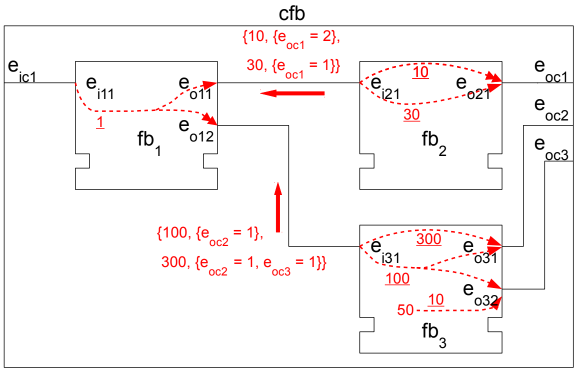


Figure 3: CFB analysis example.

propagation. In a similar way we gather information for execution paths starting from e_{o12}.

After we have gathered the information for all execution paths starting with both e_{o11} and e_{o12}, we make all combinations of the gathered data. We do this to cover all possible execution paths that can occur by generating events on both ports. Because there are two events generated at e_{o12} we must also multiply the WCET *values and number of generated events in the information gathered for that port by two. To finish the analysis for input port e_{i11} of fb_1 we add the* WCET *value of the data entry for this port to each entry in the data set obtained by generating the combinations. This is then propagated back to the event port e_{ic1} of the composite.*

The periodic execution data for the cfb is calculated using the information about periodic execution of its subcomponents, in this case only fb_3. Because the e_{o32} event output which the periodic execution of fb_3 generates is connected directly to the output of the composite, the result will include just the WCET *value of the periodic data entry (10) with its period 50, and an output to e_{oc3}.*

Assuming that we use the maximal element normalization method, which in this case does not remove any entries, the final result of the analysis is:

$$\begin{aligned}\text{WCET}(cfb) = \langle \{ \{ \langle e_{ic1}, \{ \ & \langle 611, \{e_{oc1}{=}2, e_{oc2}{=}2\} \rangle, \\ & \langle 211, \{e_{oc1}{=}2, e_{oc2}{=}2, e_{oc3}{=}2\} \rangle, \\ & \langle 631, \{e_{oc1}{=}1, e_{oc2}{=}2\} \rangle, \\ & \langle 231, \{e_{oc1}{=}1, e_{oc2}{=}2, e_{oc3}{=}2\} \rangle \ \} \rangle \}, \\ \{ \langle \ 50, \{ \langle \ & 10, \{e_{oc3}{=}1\} \rangle \} \rangle \} \rangle\end{aligned}$$

The supremum normalization would instead give:

$$\text{WCET}(cfb) = \langle \{ \{ \langle e_{ic1}, \{ \ \langle 631, \{e_{oc1}{=}2, e_{oc2}{=}2, e_{oc3}{=}2\} \rangle \ \} \rangle \}, \ \{ \langle \ 50, \{ \langle \ 10, \{e_{oc3}{=}1\} \rangle \} \rangle \} \rangle$$

3.4.1 The CFB Analysis Algorithm

The start of the composite analysis is given in Algorithm 3, where we can see separate execution of event based and periodical execution analysis and the combination of the two results into a final WCET data set.

Algorithm 3 CFBANALYSIS(cfb)

```
1: $\langle fbi, fbn \rangle \leftarrow cfb$
2: $W_e \leftarrow$ EVENTANALYSIS($cfb$)
3: $W_p \leftarrow$ PERIODANALYSIS($fbn$)
4: return $\langle W_e, W_p \rangle$
```

3.4.2 *The CFB Event Analysis Algorithm*

Algorithm 4 EVENTANALYSIS(*cfb*)

```
1: $W_e \leftarrow \emptyset$
2: $\langle\langle E_i, E_o\rangle, fbn\rangle \leftarrow cfb$
3: for each $e_i \in E_i$ do
4:   $W \leftarrow$ FBNANALYSIS$(e_i)$
5:   $W_e \leftarrow W_e \cup \langle e_i, W\rangle$
6: end for
7: return $W_e$
```

Algorithm 4 shows how we start the event analysis for a composite function block. In the *for* loop starting on line 3, we iterate through all event inputs of the composite. We start the network analysis algorithm (given in Algorithm 5) for each input and store the results of the analysis to the event WCET data set, linking it with the starting event input.

3.4.3 *The FBN Analysis Algorithm*

Before we show the algorithm for periodic execution analysis, we will first show the function block network analysis algorithm, as some concepts from the latter algorithm are basis for parts of the periodic execution analysis.

The function block network analysis described in Algorithm 5 starts from an input event port e of the composite or an output port of a function block instance and gathers WCET data for all execution paths that can be taken from that event port. The algorithm starts with initialization of the WCET data entry set W which will hold the results associated with the port e.

On line 3 we test if there is any connection leading out from the selected event port. If not, the resulting WCET data for this event is empty. Otherwise, the analysis continues at the destination port of the connection.

On line 7, we test if the destination port is an output port of the composite. If it is, we return a WCET data entry with WCET value 0 and a single output to the destination port as the only WCET data entry for the currently analyzed event.

If the connection's destination port is an input event port of a function block, we continue by first retrieving the WCET data for that function block, as shown on line 13.

As the function block WCET data can have multiple data entries (for multiple internal execution paths), we continue the analysis for each data entry separately by a *for* loop on line 15. On line 16 we initialize a temporary set W' which we will use to collect the intermediate results. The intermediate results will be added to the final result set on line 23.

With a recursive call of network analysis for each output event in the output information of the current WCET entry we gather information for execution paths started by these events, as shown on lines 17 and 18. The results of a single recursive call are stored in the W_r set, which is multiplied by the number of occurrences of the output event on line 19. On line 20 we construct all possible combinations of WCET data for execution paths started by the current output event (W_r) with the ones already gathered in the temporary set W', and use these combinations as our new temporary set. By this we have created all possible execution paths that can be taken by generating all output events of the currently examined WCET data entry.

Once we have collected the data for all possible execution paths we add the WCET value of the current data entry to all the data in the temporary set W' in line 22. We can now add this temporary data set to our final data set W, as can be seen in line 23. The final analysis results are normalized in line 25.

Algorithm 5 FBNANALYSIS(e, *fbn*)

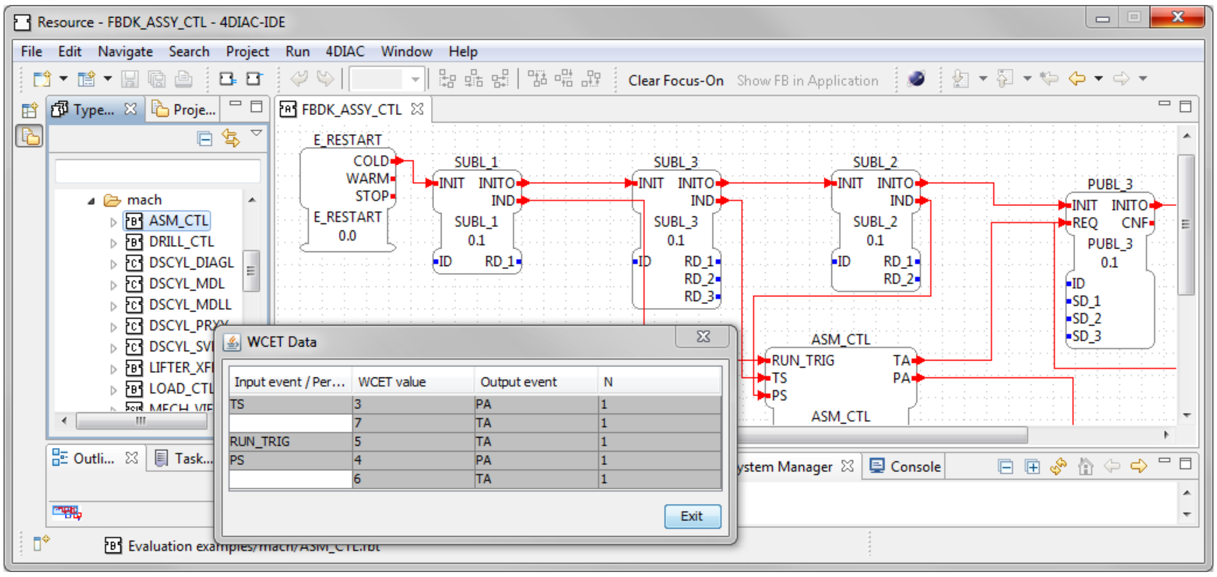
```
1: $W \leftarrow \emptyset$
2: $\langle F_i, C\rangle \leftarrow fbn$
3: if $\neg\exists\langle e_s, e_d\rangle \in C : e_s = e$ then
4:   return $\emptyset$
5: else
6:   Let $\langle e_s, e_d\rangle \in C$ be the connection for which $e_s = e$
7:   if $e_d$ is an output port then
8:     $o \leftarrow inc(\emptyset, e_d, 1)$
9:     $W \leftarrow \langle 0, o\rangle$
10:    return $W$
11:  end if
12:  Let $f$ be the FB to which $e_d$ belongs
13:  $\langle W_e, W_p\rangle \leftarrow$ WCET$(f)$
14:  Let $\langle e_i, W_t\rangle$ be the element in $W_e$ for which $e_i = e_d$
15:  for each $\langle v, o\rangle \in W_t$ do
16:    $W' \leftarrow \emptyset$
17:    for each $e_o \in E_0 : o(e_o) > 0$ do
18:      $W_r \leftarrow$ FBNANALYSIS$(e_0)$
19:      $W_r \leftarrow W_r * o(e_o)$
20:      $W' \leftarrow W' \otimes W_r$
21:    end for
22:    $W' \leftarrow \{\langle v' + v, o'\rangle : \langle v', o'\rangle \in W'\}$
23:    $W \leftarrow W \cup W'$
24:  end for
25:  $W \leftarrow$ NORMALIZE$(W)$
26:  return $W$
27: end if
```

3.4.4 *The Periodical Execution Analysis Algorithm*

Analysis of WCET for periodical execution inside a function block network is described by Algorithm 6. The algo-

Algorithm 6 PERIODANALYSIS(*fbn*)

```
1: $W'_p \leftarrow \emptyset$
2: $\langle F, C\rangle \leftarrow fbn$
3: for each $f \in F$ do
4:   $\langle W_e, W_p\rangle \leftarrow$ WCET$(f)$
5:   for each $\langle p, W\rangle \in W_p$ do
6:     $W' \leftarrow \emptyset$
7:     for each $\langle v, o\rangle \in W$ do
8:       $W'' \leftarrow \emptyset$
9:       for each $e_o \in E_o : o(e_o) > 0$ do
10:        $W_r \leftarrow$ FBNANALYSIS$(e_o,\ fbn)$
11:        $W_r \leftarrow W_r * o(e_o)$
12:        $W'' \leftarrow W'' \times W_r$
13:      end for
14:      $W'' \leftarrow \{\langle v'' + v, o''\rangle : \langle v'', o''\rangle \in W''\}$
15:      $W' \leftarrow W' \cup W''$
16:    end for
17:    $W' \leftarrow$ NORMALIZE$(W')$
18:    $W'_p \leftarrow W'_p \cup \langle p, W'\rangle$
19:  end for
20: end for
21: return $W'_p$
```



Input event / Per...	WCET value	Output event	N
TS	3	PA	1
	7	TA	1
RUN_TRIG	5	TA	1
PS	4	PA	1
	6	TA	1

Figure 4: A screenshot of the 4DIAC tool showing analysis results.

rithm starts with preparing an empty set W'_p which will hold our result.

Using the *for* loops on lines 3 and 5 we iterate through all function blocks in the network, and their period WCET data if it is defined. For each such data entry we create an temporary empty data set W'.

Lines 7 to 16 contain the same recursive analysis of all possible execution paths that can be started using the currently analyzed period WCET entry as we have already described in the function block network analysis. After the results collected in the temporary data set W' are normalized on line 17, we add them to the final result data set on line 18. They are linked to the period of execution defined for the period WCET data entry that was the origin of execution paths collected in the temporary data set.

4. IMPLEMENTATION

We have built an implementation of our analysis as a plug-in for 4DIAC-IDE [11] – an open-source tool for modeling IEC 61499 systems. The implementation includes all algorithms defined in Section 3, including both normalization methods described in Section 3.2. As the 4DIAC-IDE is an open-source tool, and built on top of the Eclipse framework, it was suitable for extending with our analysis. Integration with the 4DIAC-IDE allowed us to reuse the implementation of IEC 61499 modeling elements, graphical editors and mechanisms for loading and storing models. It also enables user to more easily use our analysis tool as it can be used on existing 4DIAC systems without the need of any model transformation. Apart from the analysis algorithms, we have also implemented GUI elements for presentation and editing of WCET data.

When starting the analysis a user can choose which data normalization method will be used during analysis. Also, user can force redoing analysis for all levels of hierarchy instead of using stored data.

A screenshot of the 4DIAC tool containing a dialog window for presentation of WCET data can be seen in Figure 4.

The Eclipse plug-in containing the implementation and a video presentation can be found at the ASSIST project web page[1].

5. EXPERIMENTS

To evaluate the applicability of our analysis approach and compare the two proposed data normalization methods we have conducted a series of experiments using our prototype analysis tool. The experiments have been carried out on four different IEC 61499 applications taken from the example systems provided by the 4DIAC-IDE [11] and FBDK [3] tools. Because true WCET values for algorithms and internal executions of service interface function blocks were not attainable, we have conducted our experiments using input sets of random WCET values, uniformly distributed between 1 and 100. To account for the randomness of the data we have repeated the experiment 1000 times for each system, each time with a new random set. For each analysis invocation we cleared the existing WCET data for all function blocks, forcing the re-analysis of all function blocks.

Information about the number of function block types, instances, hierarchical levels and average analysis running time for the systems we used in the experiment is given in Table 1. The actual analysis running time values were measured for ten consecutive invocations. To eliminate the impact of loading and storing data from and to function block models we have used temporary memory caches for each analysis invocation.

As can be seen in Table 1, the time needed for running of the analysis was very low, also for fairly complex Boiler system. The given numbers were attained using the supremum normalization, but the analysis time for the two normalization methods did not shown any significant difference.

We have also stored the resulting WCET values for all origins of execution in the analyzed system, i.e. for the individual event sources within the systems. For each origin we have compared the data attained by using the two normalization methods for the same random set of algorithm

[1]http://www.idt.mdh.se/~jcn01/research/assist/

Table 1: Experiment setup and runtime results

System	FB types	FB instances	Hierarchy levels	Average supremum running time (ms)
Boiler	60	158	6	11.8
DSCY_MDLL	20	41	4	4.6
ASSY_CTL	7	7	2	1.4
XFER_MDL	8	17	2	7.0

WCET values. The results of comparing the WCET values for the two normalization methods are shown in Table 2. They are aggregated by the systems and the execution origins, Columns one and two show the system and execution origin of the values. The third column contains the overestimation produced by the supremum method compared to the maximal elements method, on average over all random WCET input sets. In the fourth column we show the maximum of all the supremum overestimations.

In the results we can see many execution origins with only one execution path, for which the supremum method naturally did not produce any overestimation compared to the maximal elements normalization. The average WCET increase when using supremum method was between 3% and 75%. Although the maximal overestimation was only 12% for one origin of execution, it went up to 258% in the worst case. The data also shows us that there is no direct correlation between the number of components, instances, hierarchy levels and execution paths, and the overestimation of the supremum method. This indicates that the effects of the normalization mostly depend on specific combinations of function blocks rather than architectural complexity.

The two normalization methods has not shown any significant difference in terms of analysis time. This result opposes our prediction of the supremum normalization resulting in faster analysis. However, these results can be explained by a combination of multiple factors: The target applications were not complex enough for the supremum method to have much effect on the running time. This is also indicated by a large amount of execution paths having the same WCET value for both normalization methods. Also, in many cases

Table 2: Experiment results per execution origin

System	Exec. origin	Exec. paths	Average WCET increase (%)	Maximal WCET increase (%)
Boiler	1 – 8	1	0	0
	9	36	10	21
DSCY_MDLL	1 – 7	1	0	0
	8	6	3	12
ASSY_CTL	1 – 4	1	0	0
	5	3	24	80
	6	2	26	80
XFER_MDL	1, 2, 3	1	0	0
	4, 5, 6	4	75	244
	7, 8, 9	4	75	258
	10	4	75	238

the maximal elements method greatly reduced the number of examined execution paths, and thus also reducing the analysis running time. Lastly, the implementation of the supremum normalization results in a more complex algorithm than the maximal elements one, which can lead to a decline in performance in systems with a small number of hierarchical levels and execution paths. Still, we still believe that the supremum method could be useful for dealing with combinatorial complexity in very large systems.

6. RELATED WORK

An overview of the problem of WCET analysis and some of the existing methods and tools can be found in work by Wilhelm et al. [14], while some of the benefits and challenges of early mode-level WCET analysis are given by Lisper [8].

The analysis method presented in this paper has partly been based on a timing analysis approach for the ProCom component model [1]. Although we applied some of the ideas of this approach, it deals with many ProCom-specific constructs and could not be directly applied to the IEC 61499 standard. The two layered component model does not allow for non-deterministic execution on its lower level, so the method relates to using only supremum normalization in our analysis.

An approach for achieving better results when reusing WCET data for components is presented by Fredriksson et al [2]. The authors aim to reduce pessimism of compositional WCET analysis by introducing separate WCET values for clusters of different input data. Selection of appropriate WCET values is done by manually creating usage scenarios for components before the analysis. While we also define different WCET data based on event inputs of function blocks, the WCET values in their approach are clustered using values of data inputs. Our WCET data entries also include information about generated outputs, so we do not need to manually define usage scenarios. Applying the ideas of data-dependant WCET values and clustering of similar WCET data to our analysis method would be an interesting future work.

A method for verification of functional and non-functional properties in IEC 61499 is proposed by Preuße and Hanisch [9]. The authors use Symbolic Timing Diagrams and Safety-Oriented Technical Language for specification of desired behavior. Formal system model are derived by simulation. The specifications and the formal model are then used to verify the system behavior by model-checking. This approach requires various models to define a system, while our method uses only the models defined by the IEC 61499 standard. Also, it is not compositional and does not allow reuse of analysis results.

Kuo et al. [7] describe a method for worst-case reaction time analysis for IEC 61499. The method relies on compilation of systems to C code, tagging the code with timing information and then using a model checker to iterate through all possible state of the system. Besides the fact that this work determines reaction time, which is a different concept from the execution time used in our approach (see Section 3), the analysis described in the work is not performed on the model level. Systems must be fully implemented for C code to be generated, so the method does not support analysis in the early stages of development. It is also not possible to perform analysis on one hierarchical level at a time.

An approach for schedulability analysis for IEC 61499 has been presented by Khalgui et al [6]. Here, a system is first transformed to a set of tasks, each with a WCET value and a set of its predecessors and successors. The task chains acquired by the transformation can then be checked if they meet end to end deadlines set to them.

7. CONCLUSION

In this work we have presented a method for timing analysis of IEC 61499 systems. Our approach uses the standard's software models to determine WCET information for function blocks and applications. We calculate WCET data for one function block at a time, using only one level of hierarchy, by combining the WCET data of its subcomponents. The resulting data contains not only WCET values, but also information about outputs generated by the execution, and can describe multiple execution paths. In this way the WCET data is context-independent and can be stored and reused together with the function block. Our analysis method is supported by a prototype tool built on top of the 4DIAC development environment.

We have also presented two methods for data normalization that can be used to remove redundant data or reduce the size of the data sets by introducing overestimations. Data normalization, coupled with the compositional analysis approach and reuse of analysis results improves the scalability of the analysis method. To evaluate data normalization methods and viability of the analysis method presented in this paper we have conducted experiments on multiple systems obtained from example libraries of two IEC 61499 development tools.

As future work we would like to a apply the analysis to a set of systems with known WCET values, which would allow us to accurately determine the precision of our method.

Another possibility of future work is to conduct new experiments on more complex systems in order to attain better knowledge of how the different normalization methods affect the accuracy and performance of the analysis. This could lead to creation of hybrids of the two methods presented in this paper, which would, for example, create supremums of only groups of similar data.

8. ACKNOWLEDGMENTS

This work has been performed in the ASSIST project at Mälardalen University, funded by the ABB Software Research Grant Program.

9. REFERENCES

[1] J. Carlson. Timing analysis of component-based embedded systems. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering.* ACM, June 2012.

[2] J. Fredriksson, T. Nolte, M. Nolin, and H. Schmidt. Contract-based reusableworst-case execution time estimate. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 39–46. IEEE, 2007.

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

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

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

[6] M. Khalgui, X. Rebeuf, and F. Simonot-Lion. A tolerant temporal validation of components based applications. In *12th IFAC International Conference on Information Control Problems in Manufacturing (INCOM 06)*, 2006.

[7] M. Kuo, L. H. Yoong, S. Andalam, and P. Roop. Determining the worst-case reaction time of IEC 61499 function blocks. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 1104 –1109, july 2010.

[8] B. Lisper. Trends in timing analysis. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems, IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES)*, volume 225, pages 85–94. Springer, 2006.

[9] S. Preuße and H.-M. Hanisch. Verifying functional and non-functional properties of manufacturing control systems. In *Dependable Control of Discrete Systems (DCDS), 2011 3rd International Workshop on*, pages 41–46. IEEE, 2011.

[10] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2000.

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

[12] G. Čengić and K. Åkesson. On Formal Analysis of IEC 61499 Applications, Part A: Modeling. *Industrial Informatics, IEEE Transactions on*, 6(2):136 –144, may 2010.

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

[14] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

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