Handling Cyclic Execution Paths in Timing Analysis of Component-based Software

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

***Abstract*—Usage of model-driven and component-based development approaches in embedded systems allows timing analysis to be performed using system models. One of the problems rarely addressed by model-level analysis is support for analysis of cyclic execution paths. In this paper we present a method which allows compositional worst-case execution time analysis to be performed on software models containing such cycles. Our method allows defining cycle bounds for components and connections, and provides an algorithm to analyze cyclic paths containing such bounds. Additionally, we provide a possibility to propagate cycle bound definitions through the component hierarchy. The method is applied to the IEC 61499 component model and its applicability has been tested using a prototype tool.**

I. Introduction

When developing systems that operate in real-time, satisfying timing requirements is as essential as providing the correct system functionality. Many of the methods for verifying if the timing requirements of a system are met are based on analysis of worst-case execution time (WCET). Analysis of WCET on the level of source code is an already well established area [1]. However, in the embedded system domain there is an increasing trend of applying Model-Driven Engineering [2] and Component-Based Software Engineering [3], [4] development methods. In addition to other benefits, these approaches provide a possibility to perform WCET analysis on the level of system models, rather than executable code. Model-level analysis can be performed early in the development process, and can be more efficient than code-level analysis as it is performed on a more abstract view of a system. However, exploiting the full potential of model-level WCET analysis in component-based systems is not trivial.

One of the problems often neglected in model-level WCET analysis methods is support for systems containing cyclic execution paths. Such paths in software models can, for example, be used to implement functionalities such as aggregation of sensor data, or iterative algorithms. In component-based systems, cyclic paths introduces problems which are not present in analysis of standard program code loops. As opposed to program loops, which are created by loop definition keywords, cycles in component-based models are formed by a configuration of loosely bound components, having no explicit beginning or end. They can form complex patters, having multiple parallel execution paths, and can span over more than one level of component hierarchy.

In this paper we present an extension to model-level WCET analysis for component-based systems which allows the analysis to be performed on software models containing cyclic execution paths. This is achieved by multiple contributions: (i) allowing specification of cycle bounds in component-based software models, (ii) providing a method for WCET analysis of bounded cycles, and (iii) enabling the cycle bounds to be propagated through the component hierarchy. Our approach is applied to software function blocks of the IEC 61499 standard [5], and a prototype tool implemented as a plug-in to the 4DIAC development environment [6]. For the purpose of validation of the cycle analysis we have tested it on a number of scenarios, each covering a part of the desirable analysis behavior.

II. Background

In this section gives an overview of the IEC 61499 software model and the model level WCET analysis method that is extend in this work, limiting the description of both to only the information needed for understanding the work presented in this paper.

A. IEC 61499 software model

The main elements of the IEC 61499 [5] software model are *function blocks*, which are the *software components* of the standard. Although there are three types of function blocks, all of them provide the same type of interface.

The *function block interface* defines how the functionality of a function block is presented to the rest of the system. An interface consists of input and output ports, which are explicitly separated into event ports used for specifying execution flow, and data ports used for exchange of data between function blocks. Since the analysis we will describe only takes into account execution flow, while disregarding the data exchanged by the function blocks, we will present all examples without data ports and data connections. Figure 1 shows an example of a function block interface containing event input ports e_{ic1}, and event outputs e_{oc1}, e_{oc2} and e_{oc3}. Because all elements of the cycle analysis presented in this paper can be described using composite function blocks, we will only provide a description of this function block type.

Using connections between event and data ports, function block can be composed into *function block networks*. Such networks are used as implementation for *composite function blocks*. Each composite consists of an internal function block network which is connected to the ports in the interface of the composite. In this way the functionality of the network can

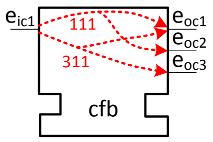

Fig. 1. Interface of the composite function block *cfb*. The red dashed arrows represent the WCET data, and are not a part of the IEC 61499 standard.

encapsulated and used as a function block in new networks, resulting in a hierarchical composition of function blocks. An example of a composite function block can be seen in Figure 2.

B. Compositional model level WCET analysis

In this section we describe some of the elements of a method for compositional model level WCET analysis of IEC 61499 systems [7]. The analysis is performed for each component in isolation, in a bottom-up manner. The analysis result for a component is obtained by composing WCET data of its sub-components. This result is then stored with the component and can be used on the next level of hierarchy, when the component is used in a composite or an application.

The base element on which the compositional analysis method is built is the context-independent WCET data for components. The WCET data consists of multiple entries, each describing an execution alternative inside the component. Each entry consists of an input event port, and the WCET value and number of generated output events that a single activation of that input event port can result in. A graphical representation of function block WCET data is shown in Figure 2, expressed as red dashed arrows. The arrow denotes that an event at the input port that the arrow starts from can generate an event at the output port that the arrow points to, and the red number next to the arrow represents the WCET value of the execution alternative. In the figure we can see that an event at the input port e_{i11} of fb_1 triggers execution that will result with one event to both e_{o11} and e_{o12} output ports, with WCET value 1. Input event e_{i31} of fb_3 on the other hand has two execution alternatives. One has WCET value 100 and results in an output event at e_{o31}, while the other has WCET value 300 and generates an event at e_{o32}.

Analysis of a composite component is based on the analysis of the internal component network of the composite. This analysis is performed by finding all execution alternatives for all its input event ports. The execution paths through the network can be determined by following the event connections between components, and by using the execution alternatives defined in the component WCET data. For each execution path, the WCET values of the utilized function block WCET alternatives are accumulated. If an execution path reaches an output event port of the composite, the number of occurrences of that event in the resulting output information is increased. We can describe the analysis of composites on the example shown in Figure 2. The analysis starts at the input event port e_{ic1}. 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_3. The result of the analysis is shown in Figure 1.

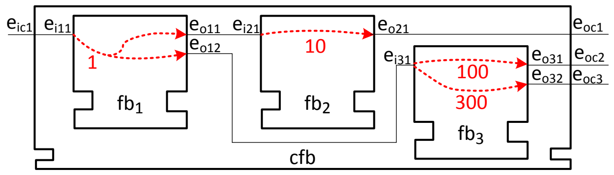

Fig. 2. An example of a composite function block. The red dashed arrows and numeric values next to them represent WCET data, and are not a part of the IEC 61499 standard.

III. CYCLE BOUND DEFINITION

Performing WCET analysis on code containing program loops is possible if the number of loop iterations has an upper limit – a loop bound. To allow for analysis of cyclic paths in component-based models we will use a similar approach and introduce cycle bounds to the model. We define cycle bounds by annotating elements of the component model. These annotations will be used by the WCET analysis to determine the maximum number of cycle iterations when the annotated element is part of a cycle. When developing composites, cycle bounds can also be defined on elements which are not contained by a cycle. Although in this case the bound will not be used when analyzing the composite in isolation, it may be used when a cycle is formed on the higher level of hierarchy. In our approach we can define two different types of cycle bounds: *component cycle bounds* and *connection cycle bounds*.

Component cycle bounds are defined by the component developer as annotations to the component interface. They are used to describe internal mechanisms that a component implements to limit the number of cyclic iterations. Because the iteration limit is a result of component's internals, it is independent of the context that the component is used in, and can be reused together with the component. Each component cycle bound is defined between one input and one output event port. The value of the bound represents the maximum number of times an execution started at the input port will result in an event at the output port if the two are used in a cycle.

When coupled with WCET data definition for components, component cycle bounds can give special semantics to WCET execution alternatives of the data. If an execution alternative starts with the input of the bound, and produces an event at the output port of the bound, during the analysis it will be treated as a cycle-forming alternative. Cycle-forming alternatives should be considered on each cycle iteration. If an execution alternative contains only the input port of the bound, but does not produce an event at the output port of the bound, it will be treated as an exit alternative. Exit alternatives should be considered only after the last iteration of a cycle.

Connection cycle bounds are defined by component integrators as annotations to connections between components. These bounds denote that the number of cyclic iterations is limited by interaction of multiple components. Connection cycle bounds can only be defined for connections that transfer control flow between components (event connections in case of IEC 61499 standard). The values of a connection cycle bound represents the maximum number of times the connection will be traversed if it is a part of a cycle.

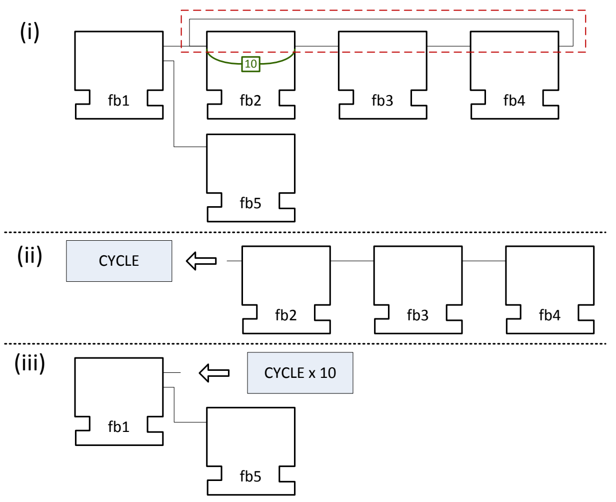

Fig. 3. The cycle analysis approach depicted on an example system. A component cycle bound is shown between two ports of *fb2* components, having bound value 10.

IV. Cycle analysis

With the ability to define cycle bound annotations on model elements, and thus defining a limit to the number of cycle iterations, we can extend the standard WCET analysis algorithms to allow analysis of systems containing cycles. The overall approach is depicted on an example in Figure 3. Cycle analysis consists of three separate stages: (i) cycle discovery, (ii) isolated cycle analysis, and (iii) merging of cycle analysis results with the results of the standard WCET analysis. Details of these three stages will be described in following sections.

A. Cycle discovery

Because cycle bounds can be defined on two different types of model elements, components and connections, the cycle discovery is also implemented in two parts of the network analysis algorithm – whenever an event connection between two components is considered, and on each usage of component WCET data. When a cycle bound definition is found during network analysis, the network is traversed in search of an analyzable cycle which contains the bound. If no such cycle is found, the the bound definition is disregarded and the cycle analysis is omitted. Such bound definition can however still be used as a candidate for hierarchical cycle bound propagation, described in Section V. If the bound is included in more than one event cycle, cycle analysis will at this point give an error, as the bound can be applied to only one of them, leaving the other unbound. In case exactly one event cycle for the bound is found, we can proceed with the isolated cycle analysis.

B. Isolated cycle analysis

The algorithm that implements the actual WCET analysis of a cycle uses a modified version of the standard network WCET analysis algorithm described in Section II-B. The network analysis is extended with a stack of currently analyzed cycles. For each cycle detected by cycle discovery, a new instance of network analysis is started form the beginning of the cycle, while adding the cycle definition to the top of the stack. By starting a new instance of analysis for each cycle, the cycle paths are isolated from the analysis performed for the rest of the component network. The cycle definition stack is used to break the connections that lead from the end of a cycle back to its beginning. Using a stack to store information about currently analyzed cycles allows starting the cycle analysis recursively, thus providing the ability to analyze multiple nested cycles. When the analysis reaches the end of the cycle that is currently at the top of the stack, the analysis for the current top-most cycle is stopped, the cycle definition is remove from the stack, and the obtained results for the isolated cycle are temporarily stored. In case that during cycle analysis the cycle discovery detects a cycle which is already in the analysis stack, and is not the top-most cycle, the analysis will report an error. In this way we detect combinations of cycles which would result in infinite recursions.

C. Merging cycle analysis results

Once the isolated analysis of a cycle is finished, the obtained results can be merged with the standard network analysis. The cycle results are first multiplied by the value of the cycle bound. If the analyzed cycle bound was defined for a component, the standard network analysis is continued using cycle exit alternatives of the WCET data of the component, i.e. alternatives that do not produce outputs to the output port of the component cycle bound. In case that the analyzed bound is defined for a connection, no additional analysis is performed.

The multiplied results, together with possible results for exit alternatives, are then added to the cumulative results of the network. The standard analysis of the execution paths covered by the cycle bound is skipped.

D. Example

We demonstrate our analysis method on an example composite which is used to filter sensor input noise by providing a mean value out of ten sensor readings. The composite and its internal component network is shown in Figure 4. While explaining the analysis of the composite we will refer to the intermediate and final results of the analysis shown in Table I.

The analysis starts from the REQ port of the composite, and following the execution path collects the temporary WCET value of 17 and one output to TMP for the initial execution of the three components, shown in Table I as Step 1.

When the standard analysis algorithm arrives to the ADD port the cycle discovery algorithm detects the cycle bound with value 9 between this port and the port $NEXT$. The discovery algorithm then performs a test to determine if the bound is

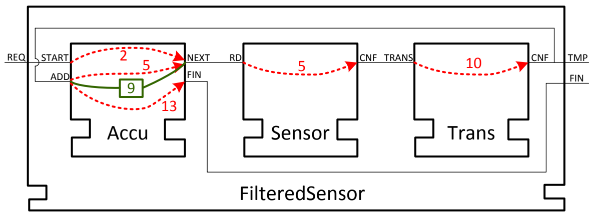

Fig. 4. Composite component used in the example of cycle analysis. Data ports and connections are not shown as the analysis disregards them.

TABLE I. INTERMEDIATE AND FINAL RESULTS OF THE ANALYSIS EXAMPLE.

Step	Analysis/result type	WCET	Outputs
1	Standard	17	TMP = 1
2	Isolated cycle	20	TMP = 1
3	Multiplied cycle	180	TMP = 9
4	Cycle exit	13	FIN = 1
5	Final (1+3+4)	210	TMP = 10, FIN = 1

a part of an analyzable cycle. Since it is, the isolated cycle analysis is triggered.

The isolated cycle analysis starts by adding the cycle bound to the cycle analysis stack, and proceeds with analysis of the network starting from ADD port, using the cycle-forming execution alternative in $Accu$ with WCET value 5. The cycle execution path is traced through all three components, collecting the cycle WCET value of 20 and one output to TMP. At this point the analysis reaches the ADD port again, and because the cycle bound for that port is on top of cycle analysis stack, the isolated cycle analysis is stopped. The results of this analysis are shown in Table I as Step 2. After the isolated analysis of the cycle is finished, the results are multiplied with the value of the cycle bound, resulting in the values shown as Step 3 in Table I.

The analysis continues with the cycle-exit alternative of the AND port. It results in WCET value 13 and an output to the FIN port, also shown as Step 4 in Table I. The multiplied cycle results are added to the cycle-exit results and combined with the temporary WCET result. The final results for the composite are shown in Table I as Step 5.

V. HIERARCHICAL PROPAGATION OF CYCLE BOUNDS

The WCET analysis method we extend in this work takes advantage of the component-based development approach, and performs the analysis in a compositional manner. Analysis of each component is performed in isolation, and only on one hierarchical level. Analysis results for a component are stored with the component, and reused together with it. However, cyclic execution paths can span over multiple levels of hierarchy. As a result, there can be a situation in which the mechanism that limits the cycle is inside of a composite component, while the actual cycle is formed outside of the composite, on a higher level of hierarchy. In this case the cycle bound will be defined on a model element that is inside of the composite, and will not be visible on the composites component's interface, which causes a problem when applying the compositional WCET analysis approach to it. To still support the compositional approach to analysis, we have to be able to represent cycle bounds defined inside composites as cycle bounds on the level of the composites' interfaces. We do this by propagation of cycle bounds.

A. Cycle bound propagation

The cycle bound propagation starts by searching the internal component network of a composite and finding bounds which are candidates for propagation. For a bound to be a candidate for propagation, it must not be a part of an event cycle inside the composite. If it is a part of an event cycle already, the bound is treated as consumed, since in this case there is no guarantee that the mechanism that implements the bound will still work if contained by a new cycle outside of the composite. However, if the bound is not a part of a cyclic path in the composite's network, the bound mechanism can be utilized when the cycle is formed on a higher level of hierarchy.

Once the propagation candidate bounds have been found, they can be tested for propagation to a pair of one input and one output port of the composite. To propagate a candidate bound, the port pair has to satisfy two requirements. First, there must be at least one path between the two ports in the internal network of the composite. Second, all paths between the port pair must traverse the candidate bound. If a combination of a port pair and a candidate bound that satisfies the two requirements is found, a new component cycle bound definition for the composite can be defined. The new bound, with the same bound value as the candidate bound, will be defined between the input and output port pair. This bound definition can then be used in any component network which contains the composite, without the need to reanalyze it.

B. Example

We will demonstrate cycle bound propagation on the example shown in Figure 5. The composite in this example is a modified version of the one used to demonstrate cycle analysis (shown in Figure 4). The new composite, shown in Figure 5 (a), leaves out the sensor component, and allows the filtering to be reused with different sensors. The sensor component can be connected on the next level of hierarchy to the S_CNF and S_RD ports on the composite interface, as shown in Figure 5 (b). To be able to still perform compositional analysis on the new system, the cycle bound defined in $Accu$ component needs to be propagated to the $Filter$ composite.

Examining the $Filter$ composite for bound propagation candidates identifies the cycle bound defined in the $Accu$ component, as it is not contained by a cycle in the internal composite network. As the candidate bound is traversed by the only path (and thus also all paths) between ports S_CNF and S_RD, it can be propagated from the $Accu$ component to the $Filter$ composite. A new bound will be defined between ports S_CNF and S_RD, and the value of the bound will be 9. The combined results of WCET analysis and bound propagation for $Filter$ are shown as annotations to its interface in Figure 5 (b).

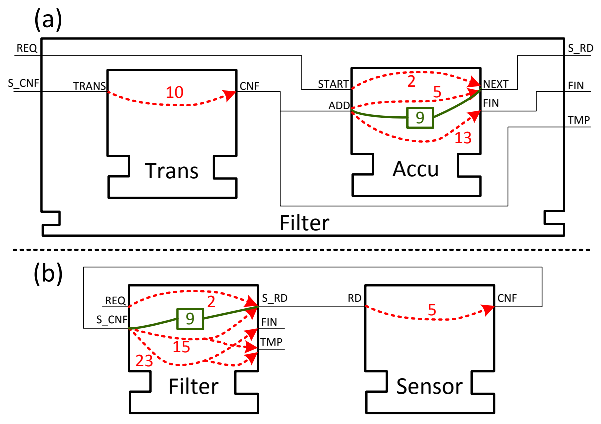

Fig. 5. Composite component used in the example of cycle bound propagation. Data ports and connections between them are omitted from the figure.

VI. Implementation and validation

The method described in this paper has been implemented as an extension to an existing prototype analysis tool (described in detail in [7]). The analysis tool is provided as a plug-in to the 4DIAC integrated development environment [6]. The 4DIAC-IDE plug-ins implementing the complete timing analysis are available for download[1].

The validity of the cycle analysis method and functionality of the prototype implementation has been tested using 16 test scenarios. The scenarios consisted of a test model and results which are expected as output of the analysis. Testing was done by re-creating the test models in the 4DIAC development environment, and applying the prototype analysis tool to them. For all test scenarios, the analysis results obtained by the prototype tool matched the expected analysis results. A detailed description is available as a technical report [8].

VII. Related work

When performing WCET analysis on code-level loops, the main problem is determining loop bounds. Examples of methods for automatic derivation of loop bounds can be seen in works by Ermedahl et al. [9], or Gustafsson et al. [10]. Although cycle bounds on model level we introduce are not a direct equivalent to loop bounds for source code, it is possible that some of these methods could be applied to derive cycle bounds from the source code of primitive components.

Vulgarakis et al. present a method [11] for analysis of resource consumption of component-based systems by combining a component model with models of component behavior. This method allows modeling of cyclic behavior contained by a single component. Compared to our method, this approach does not allow describing cyclic paths containing multiple components.

In some cases a modeling language has the ability to explicitly describe iterative execution. An example of this is *for* loop element in Simulink. Support for analysis of such constructs for Simulink can be seen in work by Kirner et al. [12]. Similar support for loop analysis is included in work by Becker et al. [13] for the Palladio Component Model. As such cyclic execution is explicit and contained in one level of hierarchy, it relates to analysis of loops on code level. Contrary to this, our approach provides a method that can be used on implicit cycles that occur as a result of component composition and can span over multiple levels of the model hierarchy.

VIII. Conclusion

In this paper we have presented a method which allows compositional timing analysis to be applied to software models that contain cyclic execution paths. The approach is applied to an existing WCET analysis for the IEC 61499 standard. We have enabled definition of cycle bounds which provide information about the upper limit of cycle iterations. Cycle bounds can be annotated to either software components or connections between them. Utilizing these bounds, we extend the standard WCET analysis with an algorithm for analysis of cyclic paths. To fully meet the needs of compositional approach to analysis, we also provide a method for propagation of cycle bound definitions through hierarchical compositions of components. The presented analysis method has been implemented as part of a prototype analysis tool, built as a plug-in to the 4DIAC integrated development environment. For the purpose of validation of the cycle analysis we have applied the tool to a set of test scenarios.

As a part of the future work, we would like to explore how source code analysis could be applied to automatically determine cycle bounds for components implemented by code, and how the overall method could be applied to component models which do not have explicit execution flow, or to analysis other than WCET.

IX. Acknowledgments

This work has been performed as part of the Ralf3 project founded by the Swedish Fundation for Strategic Research.

References

[1] 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)*, vol. 7, no. 3, p. 36, 2008.

[2] S. Burmester, H. Giese, and W. Schäfer, "Model-driven architecture for hard real-time systems: From platform independent models to code," in *Model Driven Architecture–Foundations and Applications*. Springer, 2005, pp. 25–40.

[3] I. Crnkovic and M. P. H. Larsson, *Building reliable component-based software systems*. Artech House Publishers, 2002.

[4] C. Atkinson, C. Bunse, C. Peper, and H.-G. Gross, "Component-based software development for embedded systems–an introduction," in *Component-Based Software Development for Embedded Systems*. Springer, 2005, pp. 1–7.

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

[6] 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*, july 2008, pp. 283 –288.

[7] L. Lednicki, J. Carlson, and K. Sandström, "Model Level Worst-case Execution Time Analysis for IEC 61499," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '13. ACM, 2013, pp. 169–178.

[8] L. Lednicki and J. Carlson, "Specification of tests for validation of worst-case execution time analysis for cyclic execution paths in IEC 61499," Tech. Rep., February 2014. [Online]. Available: http://www.es.mdh.se/publications/3493-

[9] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis," in *Seventh Int. Workshop on Worst-Case Execution Time Analysis, (WCET2007)*, July 2007.

[10] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, Dec 2006, pp. 57–66.

[11] A. Vulgarakis, S. Sentilles, J. Carlson, and C. Seceleanu, "Integrating behavioral descriptions into a component model for embedded systems," in *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, Sept 2010, pp. 113–118.

[12] R. Kirner, R. Lang, G. Freiberger, and P. Puschner, "Fully automatic worst-case execution time analysis for matlab/simulink models," in *14th Euromicro Conference on Real-Time Systems, 2002.*, 2002, pp. 31–40.

[13] S. Becker, H. Koziolek, and R. Reussner, "Model-Based Performance Prediction with the Palladio Component Model," in *Proc. of the 6th Int. Workshop on Software and Performance*. ACM, 2007, pp. 54–65.

[1] http://www.idt.mdh.se/˜jcn01/research/4DIAC-plugins/