

Petri Net Level WCET Analysis

Friedhelm Stappert

C-LAB

Fürstenallee 11, 33102 Paderborn

Germany

`friedhelm.stappert@c-lab.de`

Abstract

We present an approach for Worst-Case Execution Time (WCET) Analysis of embedded system software that is generated from Petri net specifications. The main characteristic of the approach is that standard Petri net analysis methods are utilized in order to automatically derive additional flow information for WCET analysis. Furthermore, the approach presented in this paper clearly separates the analysis of model behavior from the other WCET analysis phases. The method is compared with similar approaches for WCET analysis on the model level. Furthermore, an application example is presented.

1. Introduction and Related Work

A safe and precise WCET analysis must always take into account possible program flow, like loop iterations and dependencies between if-statements. This information is usually derived from the source- or object-code of a program. However, source code is often generated from higher-level specifications like StateCharts or Petri nets. In such cases, it is possible to derive additional flow information by analysing the possible behavior of the according model. Using this additional information, a more precise WCET estimation is achieved than by just analysing the generated source code alone.

To generate a WCET estimate, we usually consider a program to be processed through the phases of *program flow analysis*, *low level analysis* and *calculation*. Most WCET research groups make a similar division notationally, but sometimes integrate two or more of the phases into a single algorithm.

The program flow analysis phase determines possible program flows, and provides information about which functions get called, how many times loops iterate, if there are dependencies between `if`-statements,

etc. The information can be obtained by *manual annotations* (integrated in the programming language [15] or provided separately [3, 9, 16]). The flow information can also be derived using *automatic flow analysis* methods [5, 11, 12, 17]. Most approaches for automatic flow analysis are based on the source- or object code of a program. In contrast, this paper presents an approach based on Petri nets, thereby extending the program flow analysis phase from the source-code level to the model level.

The low-level analysis phase determines the execution time for each atomic unit of flow (e.g. an instruction or a basic block), given the architecture and features of the target system. Low-level analysis takes into account performance enhancing features like caches, branch predictors and pipelines.

In the calculation phase a program WCET estimate is computed, combining the information derived in the program flow and low-level analysis phases.

In recent work, WCET analysis on the model level has been considered for the case of StateCharts [8, 7] and Matlab/Simulink models [14]. Previous work also includes analysis on the algorithm level for the case of an MPEG decoder [1]. In the latter, knowledge about the algorithm performed by the given code – namely decoding an MPEG stream – and its possible input is exploited in order to achieve better results for the estimation of the WCET of the code. The WCET analysis of Matlab/Simulink models presented by Kirner et al. basically works by generating `wcetC` [13], a special form of C with additional annotations suitable for WCET analysis. These annotations include e.g. loop bounds that can be easily derived from the Matlab/Simulink specification. For each block of the Matlab/Simulink model, the generated `wcetC` code is analysed by an existing WCET analysis tool. The results of the analysis – namely the calculated worst-case execution times of single blocks and tasks – are then propagated back into the high-level representation of the model in the Mat-

lab/Simulink environment. Thus, the main contribution of the approach is to integrate a WCET tool in the overall environment of Matlab/Simulink using special annotations in the generated code in order to provide information about loop bounds and to propagate the calculated WCET values back to the according parts in the modelling environment. The analysis does however not derive additional information about the behaviour of the model itself.

The approach presented by Erpenbach in [8] works on StateCharts [10]. It is similar to the concept described above in that it also uses the code generated for single states of the StateChart model as basis for the WCET analysis on the model level. In addition, information is compiled about the maximum number of state transitions that can occur before the system becomes stable again after the triggering of an external event. The WCET of each possible state transition is derived by analysing the corresponding generated source code in isolation, using an existing WCET analyser. All possible sequences of state transitions – i.e. from the triggering of an external event to a stable state – are represented by an extended control flow graph. The final WCET is calculated by finding the longest path in this graph.

A key observation for the two approaches is that they mix the low-level analysis, i.e. the analysis of concrete execution times, with the analysis on the model level, by back-annotating these times into the corresponding model parts. However, since the low-level WCET analysis is performed on pieces of the generated code in isolation, the results of the overall WCET calculation are less precise since global timing effects reaching across the borders of these pieces cannot be considered.

The main advantage of the approach presented here is that it clearly separates the analysis of model behavior from the other WCET analysis phases. The model analysis does not use or produce information about execution *times*, but instead delivers information on the worst-case execution *count* of certain parts of the model. The analysis of concrete execution times is the task of a subsequent low-level analysis. Thus, the method presented here is independent from the implementation of the other WCET analysis phases.

2. Petri Net Analysis

The purpose of Petri net WCET analysis is to find the longest possible execution time a given Petri net needs to go from a defined start- to a defined end-marking. The overall architecture of our Petri net based WCET analysis is depicted in Figure 1. The analysis – divided into the two phases *reachability anal-*

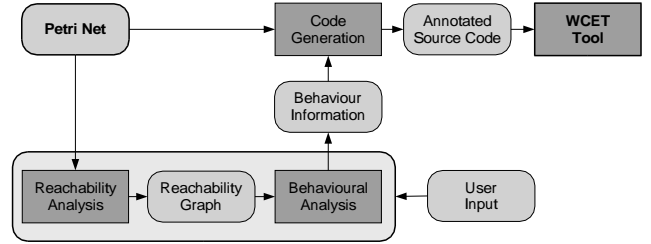


Figure 1. Petri Net Analysis Architecture

ysis and *behavioral analysis* – compiles additional information about the behavior of the net, which is handed over to an existing WCET tool [4] in form of special annotations in the generated code, using the *Flow Facts* language introduced in [3].

A detailed description of the reachability analysis and behavioral analysis is given in [18]. In this paper, we restrict ourselves to the description of the flow facts generated from the results of the Petri net analysis. The analysis derives the following information about the given Petri net:

- The worst-case number of steps the net can make until the defined end-state is reached
- For each transition, an upper bound on how often it will fire at most during all steps
- For each step, the set of transitions that could fire at that point in time

Here, a *step* is assumed to be the firing of exactly one transition.

Note that the Petri net analysis does not make any assumptions about the source-code generated from the Petri net to be analysed. Particularly, no information about execution times of single transitions is needed. Dealing with timing and source-code is done much more efficiently and precisely by the subsequently employed WCET tool. The only assumption that is made about the implementation of the Petri net execution is that it performs one transition firing per step, which is a common execution paradigm for Petri nets. Furthermore, a clear mapping between the transitions of the Petri net and their corresponding source code has to be ensured. This is achieved by integrating the source-code annotation into the code generation process as shown in Figure 1.

2.1 Example

Figure 2 shows a simple Petri net, which is a small part of a large net modelling the behavior of a Khepera minirobot. The net receives its input by means of the two input places *param* and *nextState* of transition *In*. Then, depending on the values, one of the transitions *Reset*, *Active*, *Active1*, *Active2*, *Active3* computes the new output values, which are then returned via the

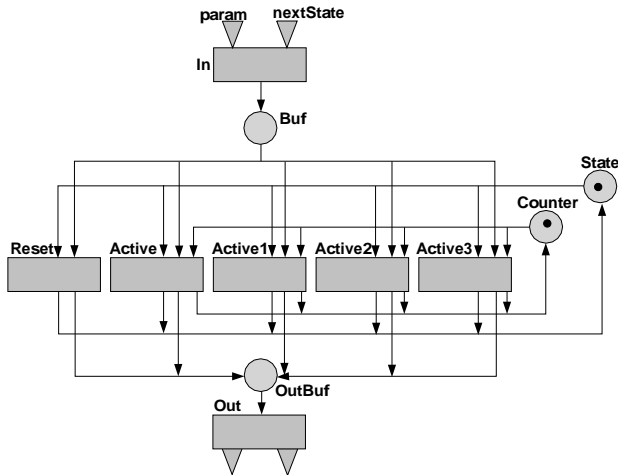


Figure 2. Example Petri Net

```

while net is alive
  get enabled transitions
  if T1 is enabled
    fire T1
  if T2 is enabled
    fire T2
  ...
end while

```

Figure 3. Petri Net Code

output places of transition *Out*. The code generated for the execution of the Petri net might look like as sketched in Figure 3.

The reachability graph resulting from the given Petri net is shown in Figure 4. As the first result, the Petri net analysis returns that the net will take at most three steps until the end-state is reached. Consequently, we know that the `while` loop in Figure 3 will take at most three iterations.

As the next result, we can see that each transition of the Petri net can fire at most once during the whole execution. Therefore, a flow fact of the form $\text{Loop}:[]:x_T \leq 1$ is generated for each transition $T \in \{In, Reset, Active, Active1, Active2, Active3, Out\}$, stating that the corresponding basic block in the generated code will be executed at most once during all iterations (denoted by the '[]' brackets) of the `while` loop (named `Loop`). For a detailed specification of the Flow Fact language we refer to [6].

Furthermore, the Petri net analysis derives detailed information about which transitions may fire in each step. As can be seen in Figure 4, only transition *In* can fire in the first step, transitions *Reset*, *Active*, *Active1*, *Active2*, *Active3* in the second step, and only transition *Out* can fire in the third step. This is reflected by the following generated flow facts:

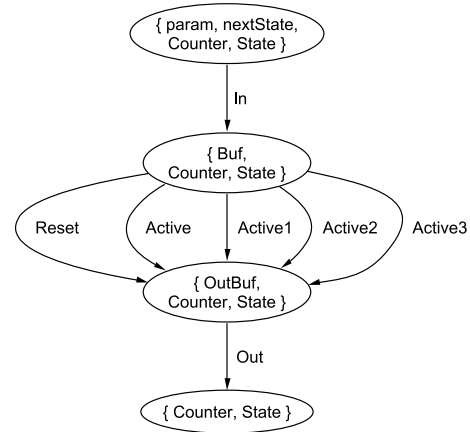


Figure 4. Reachability Graph

- $\text{Loop}:\langle 1 \rangle : x_{In} = 1$
- $\text{Loop}:\langle 1 \rangle : x_{Reset} + x_{Active} + x_{Active1} + x_{Active2} + x_{Active3} + x_{Out} = 0$
- $\text{Loop}:\langle 2 \rangle : x_{Reset} + x_{Active} + x_{Active1} + x_{Active2} + x_{Active3} = 1$
- $\text{Loop}:\langle 2 \rangle : x_{In} + x_{Out} = 0$
- $\text{Loop}:\langle 3 \rangle : x_{Out} = 1$
- $\text{Loop}:\langle 3 \rangle : x_{Reset} + x_{Active} + x_{Active1} + x_{Active2} + x_{Active3} + x_{In} = 0$

The generated code was analysed with our WCET tool prototype [4], assuming a NEC V850E as target processor [2]. The tool performs the low-level analysis and calculation as described in Section 1. The transition names in the above flow facts were manually replaced with the names of the according basic blocks in the generated code. This mapping can currently not be done automatically. However, as shown in Figure 1, annotating the source code is integrated in the code generation process. Therefore, in the final implementation the mapping will also take place without user interaction.

The results, together with the actual WCET of the code are shown in Figure 5. First, the code was analysed without consideration of the generated flow facts (column named 'no facts'). Only the mandatory upper bound for the number of loop iterations was given, since otherwise a WCET analysis would not be possible. Without flow facts, the actual WCET (rightmost column) was overestimated by about 100% (2425 versus 1177 cycles). When taking into account the flow facts (column named 'with facts'), the WCET estimation was 1744 cycles, which is significantly closer to the actual WCET. This improvement is due to the fact that the calculation phase can make less pessimistic assumptions about the possible execution paths of the code. The results of the low-level analysis are not affected.

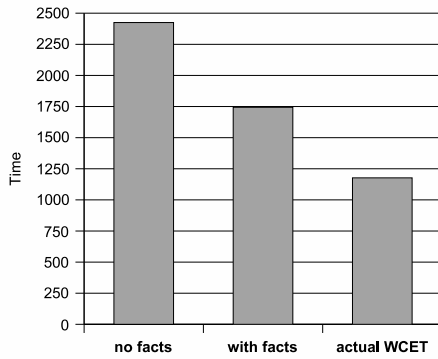


Figure 5. Analysis Results

3. Conclusion

In this paper, a WCET analysis for Petri nets was presented. Independently of the source-code generated for a given Petri net, the analysis compiles information about the behavior of the net. The gathered information is then converted to a set of flow facts, which are exploited by a subsequently employed source-code based WCET tool. Using this additional information, a more precise WCET estimation is achieved than by just analysing the generated source code alone. Information from the model (i.e. Petri net) level is therefore not lost on the next lower level of the generated source-code.

From the example in Section 2.1, it can be seen that the flow facts derived by the Petri net analysis lead to a much tighter prediction of the WCET, even for the relatively simple net presented in the example.

A basic feature of the presented approach is the strict separation from other WCET analysis phases, thereby consequently following our overall approach for a modular WCET tool architecture [4, 6]. This modular approach makes it easy to e.g. replace an algorithm for a certain analysis phase with a more efficient one.

Although the presented approach is based on Petri nets, the basic proposal of separating the model analysis from the other WCET analysis phases could also be applied for other modelling paradigms, like e.g. State-Charts or Matlab/Simulink models.

References

[1] Peter Altenbernd, Lars O. Burchard, and Friedhelm Stappert. Worst-Case Execution Times Analysis of MPEG-2 Decoding. In *Proc. 6th International EUROMICRO Conference on Real-Time Systems*, Stockholm, June 2000.

[2] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3rd edition, January 1999. Document no. U12197EJ3V0UM00.

[3] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21th*

IEEE Real-Time Systems Symposium (RTSS'00), November 2000.

[4] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems. In Paul Pettersson and Sergio Yovine, editors, *Workshop on Real-Time Tools*, Aalborg, Denmark, August 2001. Published as Technical Report No. 2001-014, Department of Information Technology, Uppsala University, Uppsala, Sweden.

[5] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[6] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Sweden, June 2003.

[7] E. Erpenbach, F. Stappert, and J. Stroop. Compilation and Timing Analysis of Statecharts Models for Embedded Systems. In *The Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*, Washington, D.C, Oct. 1999.

[8] Edwin Erpenbach. *Compilation, Worst-Case Execution Times and Schedulability Analysis of Statecharts Models*. PhD thesis, University of Paderborn, Germany, 2000.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1978.

[11] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[12] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, September 2000.

[13] Raimund Kirner. The programming language wctc. Technical Report 2/2002, Technische Universität Wien, Institut für Technische Informatik, 2002.

[14] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschnier. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *EUROMICRO Conference on Real-Time Systems*, 2002.

[15] Raimund Kirner and Peter Puschnier. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[16] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[17] F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[18] Friedhelm Stappert. *From Low-Level to Model-Based and Constructive Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2004. C-LAB Publication, Vol. 17, Shaker Verlag, ISBN 3-8322-2637-0.