Finding DU-Paths for Testing of Multi-Tasking Real-Time Systems using WCET Analysis

Daniel Sundmark, Anders Pettersson, Christer Sandberg, Andreas Ermedahl, and Henrik Thane
Department of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{daniel.sundmark,anders.pettersson,christer.sandberg,andreas.ermedahl,henrik.thane}@mdh.se

Abstract

Memory corruption is one of the most common software failures. For sequential software and multi-tasking software with synchronized data accesses, it has been shown that program faults causing memory corruption can be detected by analyzing the relations between defines and uses of variables (DU-based testing). However, such methods are insufficient in preemptive systems, since they lack the ability to detect inter-task shared variable dependencies. In this paper, we propose the use of a system level shared variable DU analysis of preemptive multi-tasking real-time software. By deriving temporal attributes of each access to shared data using WCET analysis, and combining this information with the real-time schedule information, our method also detects inter-task shared variable dependencies. The paper also describes how we extended the SWEET tool to derive these temporal attributes.

1 Introduction

Software complexity, and especially that of embedded real-time systems, is rapidly increasing. Consequently, the task of finding faults is getting more difficult. Among the most common software failures is memory corruption, e.g., out-of-bound writes, pointer failures, and usage of uninitialized variables. For multi-tasking systems, failures also encompass non-synchronized reads and writes, and non-reentrance failures. There exist several methods that address these problems, typically in terms of static define and use analysis (DU analysis), or DU-based testing methods. However, the majority of these methods only address systems with a single thread of non-preemptive execution [7, 9, 8], or multi-tasking systems with task interference restricted to synchronous interference [3, 4].

In our previous work we have addressed testing of multi-tasking real-time systems where input and output from a task is given and produced at the beginning and at the end of the task's execution respectively [15]. We relaxed the model to encompass systems where task communication could be performed within semaphore guarded critical sections [14], and later also by non-synchronized shared variables anywhere in the execution of the tasks [13]. In this paper, we extend previous results and show how to derive all DU-paths from a preemptive real-time system using WCET analysis.

```
1.  a:=b+4
2.  if expression is true then
3.      result:=a+1
4.  else then
5.      result:=a*2
```

Figure 1. Example of defs and uses.

2 Background

Classic data-flow unit testing, for single thread execution programs, tests read and write accesses to program variables [12]. Data-flow is identified in terms of definitions and uses of data, where a *definition* is an assignment of a value to a variable. A *use* is an action of reading a variable or container. One classic DU relation is the *DU-path*. A definition d (write) and a use u (read) of variable x constitutes a DU-path (d,u) if and only if there exists a control-flow path p from d to u, such that p contains no other definitions of x. E.g., in Figure 1, uses are enclosed in a selection statement, yielding the following DU-paths: $\{(\mathtt{a}^1, \mathtt{a}^3), (\mathtt{a}^1, \mathtt{a}^5)\}$. Here, `a` is the identity of the variable and the index corresponds to the line number in the code.

When testing using the all-DU-path coverage criterion [17], DU-paths define test items that should be covered. Hence, should the software contain any unintended, and erroneous, DU-paths, these will be discovered by a full all-DU-path coverage testing. Coverage (i.e., the ratio between identified test items and exer-

ECRTS 2007
7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2007/1191

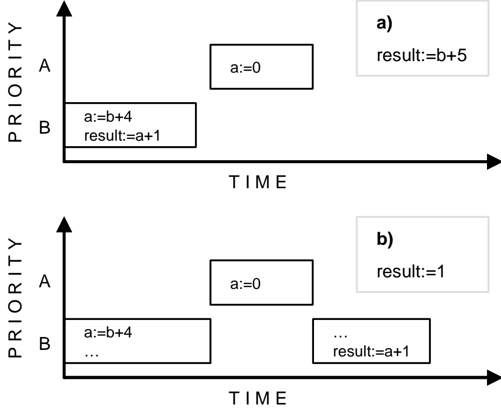


Figure 2. Shared variable communication (assuming that the conditional expression is true in task B).

cised test items) is the state-of-the-practice metric for test thoroughness. A 100% coverage describes a fully tested software with respect to a certain test criterion (e.g., branch, path, or DU-path coverage).

When moving from unit-level DU-based testing to system level testing of preemptive real-time systems, a whole new dimension of complexity is added - concurrent access to shared resources. Figure 2 shows an example execution of two concurrently executing tasks, A and B. B consists of the code from Figure 1. A has a higher priority than B, and contains a definition of variable `a`. In B there is a definition (`a:=b+4`) and two uses (`result:=a+1` and `result:=a*2`) of the same variable `a`. Hence, at task level, A only has a definition and no DU-paths. B has the following set of DU-paths: $\{(\mathtt{a}^{B1}, \mathtt{a}^{B3}), (\mathtt{a}^{B1}, \mathtt{a}^{B5})\}$, here indexed with the task identity and line number. Assume that, for correct intended behavior, B shall always complete the definition and the use in a sequence, i.e., A is not allowed to preempt B in between the definition and the use. Figure 2a illustrates the case without preemption and in-between definition of `a`. In Figure 2b, A preempts B and redefines the variable, thus corrupting the value. Using the task level DU-path sets for testing on system level will leave out scenarios as in Figure 2b where more paths evidently exist. In order to capture system level DU-paths, it is necessary to consider all scenarios (i.e., where the definition in A executes strictly before, strictly after, and where it preempts the definition and use in B). The resulting DU-paths are: $\{(\mathtt{a}^{B1}, \mathtt{a}^{B3}), (\mathtt{a}^{B1}, \mathtt{a}^{B5}), (\mathtt{a}^{A}, \mathtt{a}^{B3}), (\mathtt{a}^{A}, \mathtt{a}^{B5})\}$. In Figure 3, the scenarios from Figure 2 are revisited, but here, the focus is on the exact times when the accesses are executed. E.g., in Figure 3a definition `a:=b+4` is executed at def_1 and overwritten at rd_1. In Figure 3b,

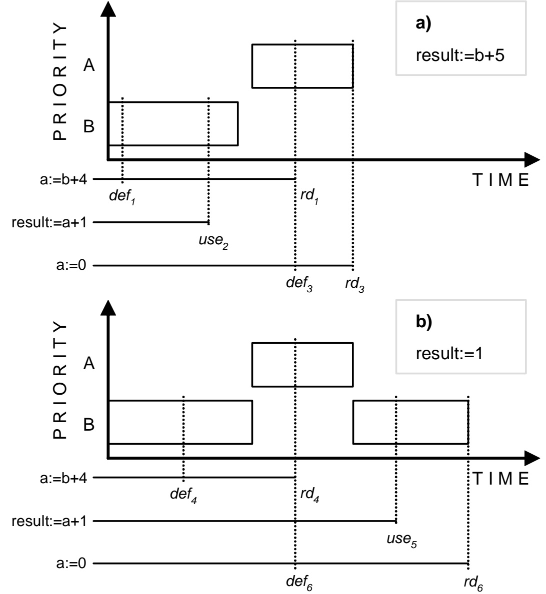


Figure 3. Shared variable access attributes.

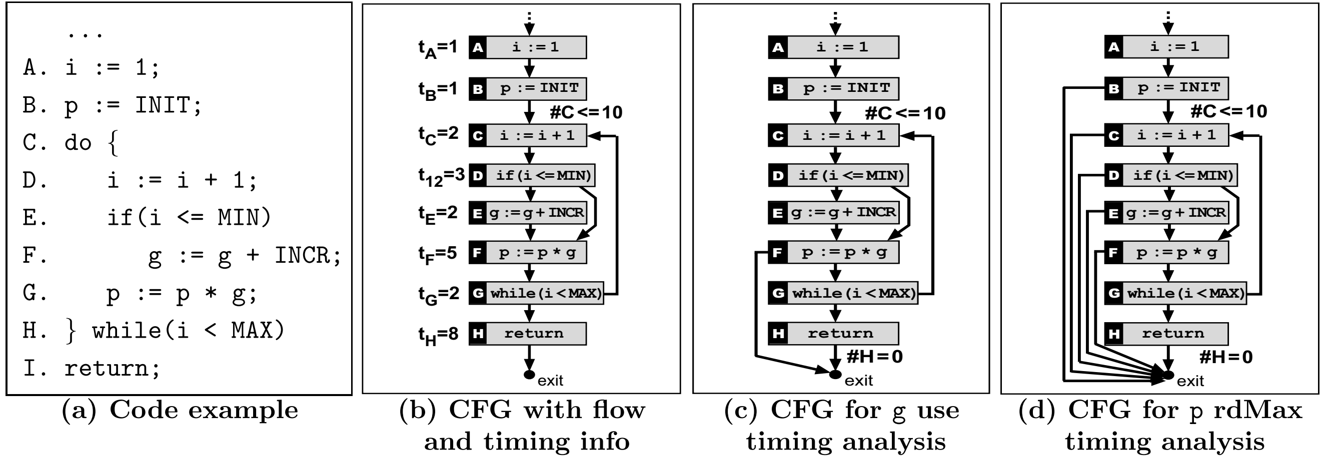
`a:=b+4` is executed at def_4 and overwritten at rd_4. Generally, for each access x, there is an interval with extremal values $x.min$ and $x.max$ within which x can be executed. Furthermore, for each definition d, there is a point in time $d.rdMax$, where d is safely overwritten. Hence, to derive feasible shared variable DU-paths on system level, we require (1) task-level information of when shared variables may be accessed by each task in the system, and (2) system-level information of how these tasks are scheduled and temporally interfere.

In this paper, we derive all system-level shared variable DU-paths by extending the method presented in [13]. The contributions of this article are:

- An extension to the SWEET WCET tool [6], able to derive task-level shared variable access times.
- The use of task-level shared variable access times for deriving system-level shared variable DU-paths.
- An experimental evaluation of the effectiveness of the approach.

We assume a uni-processor real-time system S. The operating system and application software (implemented as a set of tasks W_S) operates to control an external environment (e.g., a vehicular or industrial mechatronic control system). We assume strictly periodic tasks that follow the *single shot semantics* [2]. The tasks are scheduled using the *fixed priority scheduling* policy [1], and each task is assigned a unique priority.

In this paper, we represent a task as a 7-tuple, $\langle\, T, O, P, D, ET, \mathcal{D}, \mathcal{U}\,\rangle$, where T is the *periodicity* of the task. The task's release time for each period is calculated by adding the *offset* O to T. The scheduling



(a) Code example (b) CFG with flow and timing info (c) CFG for g use timing analysis (d) CFG for p rdMax timing analysis

Figure 4. Example of timing analysis for defs and uses.

mechanism determines which released task will execute based on the task's *priority*, P. The task's latest completion time is determined by its *deadline*, D. Further, a task encompass information regarding the best- and worst-case *execution time* of the task, ET, as well as two sets of shared data accesses, defined in terms of *definitions* ($\mathcal{D}$) and *uses* ($\mathcal{U}$) of the data. For each least common multiple of the tasks' period times (LCM), the system schedule performs a recurring pattern of task instance releases (*jobs*). In each LCM, each task can spawn one or more jobs. The release time R and deadline D of a job are calculated using the task T, O, and D properties respectively.

3 System-Level DU Analysis

In this section, we show how to derive information of when shared variables may be accessed by each task in the system (Section 3.1), and how to combine this information with the real-time schedule in order to derive all system-level DU-paths (Section 3.2).

3.1 Task-Level Analysis

The task-level analysis derives the temporal properties for each shared variable access (definition or use) in each task $w \in W_S$. Three properties (*min*, *max*, and *rdMax*) are derived for each definition, and two properties (*min* and *max*) are derived for each use. As *min* and *max* are analogous for definitions and uses, we will focus on the definition properties. Assuming a definition d that defines a variable x, the $d.min$ property describes the *shortest* possible time from the start of the task to the statement containing d. The $d.max$ property describes the *longest* possible time from the start of the task to the statement containing d. The $d.rdMax$ property describes the *longest* possible time for a path p, starting at task start s and ending at a statement e, such that d is on p, e contains a statement that redefines x, and no other redefinitions of x are made between d and e. Intuitively, this property describes the time (relative to the start of a task) where a definition d is safely overwritten.

The SWEET tool (SWEdish Execution time Tool) [5] is a research prototype WCET tool developed at Mälardalen University [10]. SWEET consists of three distinguished phases: a *flow analysis* where bounds on the number of times different entities in the code can be executed is derived, a *low-level analysis* where bounds of the execution times for instructions are derived, (taking into account the effects of pipelines and potentially instruction caches), and a final *calculation* phase where the flow and timing information is combined to yield a WCET estimate.

We have modified SWEET to, except the "normal" program WCET and BCET estimates, also produce estimates upon the above mentioned *min*, *max*, and *rdMax* values. Initially, we perform the flow- and low-level analysis of the program, but not the calculation. The result can be seen as a control-flow graph (CFG) containing both flow- and timing bounds and with two extra *start* and *exit* nodes. Figure 4(a) depicts an example code with two globals `g` and `p`. Figure 4(b) illustrates the CFG for the code. The flow analysis has derived a loop bound of 10, expressed as an upper bound on the number of times node `C` could be executed. Each node is also given a timing bound by the low-level analysis, valid each time the node is executed.

Secondly, we perform a *reaching definition* (RD) analysis for global variables [11]. The analysis derives, for each global variable, where in the program it may be used and defined as well as how far each definition may reach. Since pointers could be used to update globals, the RD takes the input of a pointer analysis.

We derive the different estimates using IPET calculation [5]. In IPET each node and/or edge in the CFG is given a time (t_{entity}), and a count variable (x_{entity}), the latter denoting the number of times that block or edge is executed. The WCET is found by maximising the sum $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows. There are, e.g., constraints specifying that the *start* and *exit* nodes each must be taken exactly once, and constraints specifying that that each node must be entered the same number of times as it is exited. The estimate is normally derived using integer linear programming (ILP). The BCET is found by minimizing the same sum, subject to the same constraints.

Our analyses start from the above mentioned graph. Depending on what timing values to derive, we modify the graph by adding extra edges and flow contraints. E.g., the graph for deriving *min*, *max* for a use u is constructed by adding en extra edge from the node holding u to the *exit* node. Additionally, for all other edges going to the *exit* node we add a flow constraint specifying that its source node cannot be taken. Thus, we force the IPET calculation to exit through our newly created exit-edge, thereby deriving the best-case and worst-case estimates for u, instead of the "normal" BCET and WCET. Figure 4(c) shows the CFG for calculating *min* and *max* for the use of `g` in node `F`. For each global use and def derived in the RD analysis, we construct a corresponding graph. The resulting graphs are given as input to SWEET to derive the corresponding *min* and *max* values.

To derive *rdMax* for a def d we first use the RD analysis to derive the set of nodes which d may reach. From these nodes we add an extra edge to the exit node. Additionally, for all other exit-edges going from a node which d cannot reach, we add a flow constraint specifying that its source node cannot be taken. Thus, we force the IPET calculation to exit through *one* of the nodes d may reach. The *rdMax* value is derived by a WCET calculation upon the resulting graph. Figure 4(d) shows the graph for calculating the *rdMax* value of the `p := INIT` definition in node `B`.

3.2 System-Level Analysis

The algorithm for deriving system-level DU-paths is based on the algorithm deriving Execution Order Graphs (defined 1999 by Thane and Hansson [15] as directed reachability graphs of all possible execution orderings from a scheduled set of task instances during a periodically repeated FPS schedule). Basically, the EOG algorithm simulates the behaviour of a real-time FPS scheduler, considering all interleaving pattern alternatives caused by task execution time variations. As an example, Figure 2 displays two different execution orderings of the same system caused by execution time variations in task B. In our analysis, we modify the EOG algorithm such that it given our extended task model (with $\mathcal{D}$ and $\mathcal{U}$ properties) instead generates all possible DU-paths of the system. The algorithm simulates the behaviour of the system by exhaustively searching all task interleaving patterns, and acting upon shared variable accesses in the tasks at various times. Throughout the analysis, each access holds a certain state (*dead*, *active* or *live*). An *active* access has been executed, or can be executed at any time until it has become *live* or *dead*. A *live* access has safely been executed, and not been safely overwritten by another access. A *dead* access is neither *active* nor *live* (i.e., the access has safely not yet been executed, is safely overwritten, or has safely passed the time where it can affect the result of the analysis). The rules for making the transitions between these access states constitute the foundation of the system-level analysis:

Definition rules:

1. At $d.min$, d makes a transition from $dead \rightarrow active$.
2. At $d.max$, d makes a transition from $active \rightarrow live$.
3. At $d.rdMax$, d makes a transition from $live \rightarrow dead$.

Use rules:

1. At $u.min$, u makes a transition from $dead \rightarrow active$.
2. At $u.max$, u makes a transition from $active \rightarrow dead$.

DU-path rules:

1. At $d.min$, all DU-paths (d, u), such that $u.var = d.var$ **and** u is currently active, are derived.
2. At $u.min$, all DU-paths (d, u), such that $u.var = d.var$ **and** d is currently live **or** active, are derived.

These seven rules are implemented in the DuAnalysis algorithm. This algorithm (Figure 5) is a slight variation[1] of the original EOG algorithm [16], built upon the manipulation of two data structures. Throughout the analysis, an abstract state of type **State** propagates through the execution of the system. For each execution of a job, the abstract state is changed according to the **Transition** created by executing the job. **State** represents the current abstract state of the execution, and contains information of currently live definitions, active definitions, active uses, and encountered DU-paths:

$$\textbf{State} \quad : \quad \{liveDefs, activeDefs, activeUses, duPaths\}$$

Transition represents a change of state incurred by the execution of a (partial) job. Thus, **Transition** contains new active definitions, killed live definitions, killed active definitions, killed active uses, and the

[1] Changes to the original algorithm are blackened in Figure 5.

```
DuAnalysis (state, transition, rdy, RI, SI)
{
        // When is the next job(s) released?
1.      t =NextRelease(SI)
2.      if rdy = ∅
3.         rdy = MakeReady(t, rdy)
4.         if rdy ≠ ∅
5.            DuAnalysis(state, transition, rdy, RI, (t, SI.r])
6.         else state = SwitchTask(transition, state, RI)
7.      else
           // Extract the highest priority job in rdy.
8.         J               = Dispatch(rdy)
9.         [α, β)          = [max(J.R, RI.l), max(J.R, RI.l) + J.WCET)
10.        a'              = α + J.BCET
11.        b'              = β
12.        state           = SwitchTask(transition, state, RI)
13.        transition      = Execute(state, J, [α, β), RI)

           // Add all lower prio jobs released before J's termination,
           // or before a high priority job is preempting J.
14.        while((t < β) ∧ (Prio(t) < J.P))
15.              rdy    = MakeReady(t, rdy)
16.              t      = NextRelease((t, SI.r])

           // Does the next scheduled job preempt J?
17.        if ((t < β) ∧ (Prio(t) > J.P))
              // Can J complete prior to the release of the next job at t?
18.           if t > a'
19.              DuAnalysis(state, transition, rdy, [a', t), [t, SI.r])
20.              if rdy = ∅
21.                 DuAnalysis(state, transition, MakeReady(t, rdy), [t, t), (t, SI.r])
22.              else if t = a'
23.                 DuAnalysis(state, transition, MakeReady(t, rdy), [t, t), (t, SI.r])

              // Add all jobs that are released at time t.
24.           rdy = MakeReady(t, rdy)

              // Best and worst case execution time prior to preemption?
25.           J.BCET = max(J.BCET − (t − (max(J.R, RI.l)), 0)
26.           J.WCET = max(J.WCET − (t − (max(J.R, RI.r)), 0)
27.           Preempt(J, (t − (max(J.R, RI.l)), (t − (max(J.R, RI.r)))
28.           DuAnalysis(state, transition, rdy ∪ {J}, [t, t], (t, SI.r])

           // No preemption.
29.        else  if t = ∞ // Have we come to the end of the analysis?
30.                 DuAnalysis(state, transition, rdy, [a', b'), [∞, ∞]) // Yes

31.          else  // More jobs to execute.
               // Is there a possibility for a high priority job to succeed
               // immediately, while low priority jobs are ready?
32.            if (rdy ≠ ∅ ∧ t = β)
33.               DuAnalysis(state, transition, MakeReady(t, rdy), [t, t), (t, SI.r])
34.               if a' ≠ b' // And one branch for the low priority job.
                     // The regular succession of the next job
35.                  DuAnalysis(state, transition, rdy, [a', b'), [t, SI.r))
}
```

Figure 5. The DuAnalysis algorithm.

WCET of the executed job:

$$\textbf{Transition} \;:\; \{newActiveDefs, killedLiveDefs, killedActiveDefs, killedActiveUses, wcet\}$$

Intuitively, the combination of a **State** a_1 and a **Transition** a_1' yields a new **State** a_2, representing the original state affected by the changes in a_1'. E.g., if a_1 contains a set of liveDefs $\{d_1, d_2, d_3\}$, and a_1' contains a set of killedLiveDefs $\{d_2\}$, then a_2's set of liveDefs will look as follows: $\{d_1, d_3\}$. In the algorithm, this process is formalized by the functions Execute and SwitchTask, where

$$\text{Execute} : \textbf{State} \times \textbf{Job} \times \textbf{Ivl} \times \textbf{Ivl} \rightarrow \textbf{Transition}$$

$$\text{SwitchTask} : \textbf{Transition} \times \textbf{State} \times \textbf{Ivl} \rightarrow \textbf{State}$$

In essence, the Execute function produces a change of abstract state (**Transition**) incurred on original abstract state by executing a certain job. The SwitchTask function produces a new abstract state (**State**), based on the original abstract state and the changes described by **Transition**. The implementation of these functions are directly based on the Definition, Use, and DU-path rules shown above. Two more structures (**Ivl** and **Job**) are used in the analysis. **Ivl** defines a time interval by its extremal values l and r. **Job** represents a task instance and contains definitions, uses, job priority, release time, BCET and WCET:

$$\textbf{Job} \;:\; \{\mathcal{D}, \mathcal{U}, P, R, BCET, WCET\}$$

Roughly, the DuAnalysis algorithm starts with an empty **State** at time 0 by scheduling the highest prioritized ready job j. Using the Execute function, j's **Transition** is derived. Next, if j is always finished before the next higher priority job is released, SwitchTask combines the **Transition** with the old **State** to a new **State**. The algorithm increments the time and schedules the next job. Else, if j is certainly preempted by a higher priority job, SwitchTask combines the **Transition** with the old **State** to a new **State** - but only regards the events that predate the preemption, stores the remainder of j, increments the time, and schedules the higher prioritized job. Else, if j *might* be preempted, the algorithm splits into two recursive branches, one of which considers the case with a preemption, and the other considers the case with no preemption. This behaviour is repeated until all jobs in the LCM are analysed. In order to derive the **Transition** created by executing a job j, the Execute function works through all shared variable accesses in j in a chronological order. Each access is treated according to its corresponding definition or use rule. SwitchTask creates a new **State** by adding the changes in j's **Transition** to the **State** prior to the execution of j. If j is not preempted, all changes in **Transition** are considered when creating the new **State**. Otherwise, only those changes prior to the preemption time are considered.

4 Evaluation

As an experimental evaluation of our method, we provide analysis results from five different multitasking real-time systems (S_1-S_5), each scheduled in three different ways (Cfg1-3). All systems comprise control-oriented code (e.g., calculation of planet orbits (S_1), a control system for a forklift able to solve the Towers Of Hanoi problem (S_2), etc.), and include intertask communication via shared variables. In Table 1, **Ts** and **GVar** refer to the number of tasks and global variables respectively. C_{DU} refers to the number of combinatorially feasible DU-paths (i.e., each definition d and use u of the same shared variable may naively

Sys	Ts	GVar	C_{DU}	Cfg1			Cfg2			Cfg3		
				F_{DU}	F_{DU} / C_{DU}	$rt(ms)$	F_{DU}	F_{DU} / C_{DU}	$rt(ms)$	F_{DU}	F_{DU} / C_{DU}	$rt(ms)$
S_1	4	18	216	155	71.6%	4282	147	68.1%	676	134	62.0%	19
S_2	4	27	183	N/A	N/A	N/A	176	96.2%	3750	163	89.1%	30
S_3	4	22	34	32	94.1%	217	32	94.1%	107	27	79.4%	4
S_4	3	7	44	34	77.3%	137	37	84.1%	24	24	54.5%	1
S_5	2	4	236	204	86.4%	422	196	83.1%	248	180	76.3%	12

Table 1. Evaluation Results.

form a DU-path (d, u)). F_{DU} refers to the number of DU-paths found feasible by the DuAnalysis algorithm, and rt refers to the analysis time (in milliseconds). Hence, 1 - (F_{DU} / C_{DU}) describes the percentage of DU-paths that safely do not have to be considered during testing. As for the configurations, Cfg3 completely separates all tasks in time and suffer no preemptions. In contrast, Cfg1 maximizes the number of task preemptions. Cfg2 is an in-between configuration of Cfg1 and Cfg3. Generally, with a less complex scheduling, more DU-paths are found infeasible (except for Cfg1 and Cfg2 of S_4). Note also that the system-level analysis of Cfg1 of S_2 proved too complex to execute. Since all other configurations finished within a few seconds, we will investigate this problem further.

5 Conclusion

For multi-tasking real-time systems, failures at system level caused by concurrently executing tasks cannot be revealed by tests at task level. In this paper, we have presented and evaluated a method that derives all possible DU-paths, enabling system-level testing of task inter-dependency failures. Our system model is restricted to a model suitable for small embedded real-time systems. In our future work we plan to show how our method can be used on a more relaxed system model, e.g., a system model based on transactions of tasks instead of strictly periodic tasks. Our method however requires a finite (periodically or non-periodically) repeated system behaviour. Further, semaphores and critical sections for shared variable access protection can be added to our method without major efforts as described in [14].

References

[1] N. C. Audsley, A. Burns, R. I. Davis, and K. W. Tindell. Fixed priority pre-emptive scheduling: A historical perspective. In *Real-Time Systems journal*, volume 8(2/3). Kluwer A.P., March/May 1995.

[2] T. Baker. Stack-based scheduling of real-time processes. In *Real-Time Systems Journal*, volume 3(1), pages 67–99, 1991.

[3] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. In *IEEE Software*, volume 8(2), pages 66–74, 1991.

[4] S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and B. S. Lee. Testing of concurrent programs based on message sequence charts. In *Proceedings IEEE International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 72–82, Vol., Iss., 1999.

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

[6] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for Infeasible Path Calculation. In *Sixth International Workshop on Worst-Case Execution Time Analysis, (WCET'2006)*, Dresden, Germany, July 2006.

[7] M. Harrold and M. Sofia. Interprocedural Data Flow Testing. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification*, pages 158–167, 1989.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.

[9] J. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. In *IEEE Transactions on Software Engineering*, volume 9(5), pages 347–354, May 1983.

[10] Mälardalen University. WCET project homepage, 2007. www.mrtc.mdh.se/projects/wcet.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2nd edition*. Springer, 2005. ISBN 3-540-65410-0.

[12] I. S. G. of Software: Engineering Terminology. IEEE Standards Collection, IEEE Std 610.12-1990. September 1990.

[13] A. Pettersson, D. Sundmark, H. Thane, and D. Nyström. Shared Data Analysis for Multi-Tasking Real-Time System Testing. In *Proceedings of Second Symposium of Industrial Embedded Systems*, July 2007.

[14] A. Pettersson and H. Thane. Testing of Multi-Tasking Real-Time Systems with Critical Sections. In *Proceedings of Ninth International Conference on Real-Time and Embedded Computing Systems amd Applications*, Tainan City, Taiwan, R.O.C, 18-20 February 2003.

[15] H. Thane and H. Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[16] H. Thane and H. Hansson. Testing Distributed Real-Time Systems. In *Journal of Microprocessors and Microsystems*, pages 463–478. Elsevier, 2001.

[17] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.