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.

1 Introduction

The complexity of software, and especially that of embedded real-time systems, is rapidly increasing. Faster CPUs allow for more functionality, both in terms of size and concurrency. Consequently, the task of finding faults is getting more complex and 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, failure examples also encompass non-synchronized reads and writes, and non-reentrance failures. There exist several methods and tools that address these problems, typically in terms of static define and use analysis (DU analysis), or DU-based testing methods. However, the majority, if not all, of these methods and tools 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 (e.g., messages, monitors or semaphores [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 of a task’s execution and at the end of the task’s execution respectively [15]. We relaxed the model to encompass testing of 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.

2 Motivation

Classic data-flow based unit level testing, for single thread execution programs, tests read and write accesses to program variables [12]. The 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: \(\{(a^1, a^3), (a^1, a^5)\}\). 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 crite-
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 exercised 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).

2.1 Problem Formulation

When moving from DU-based testing of single threads of execution to system level testing of preemptive real-time systems, a whole new dimension of complexity is added to the testing process - namely concurrent access to shared resources. In Figure 2, we show an example execution trace 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: \{a^{B1}, a^{B3}\}, indexed with the task identity and line number. Assume that, for correct intended behavior, B shall always complete both 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 with no 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: \{(a^{B1}, a^{B3}), (a^{B1}, a^{B5}), (a^A, a^{B3}), (a^A, a^{B5})\}.

In Figure 3, the same scenarios as in Figure 2 are depicted, but here, the focus is on the exact times when the accesses are carried out. E.g., in Figure 3a definition a:=b+4 is executed at def\textsubscript{1} and is overwritten at rd\textsubscript{1}. In Figure 3b, the same definition is executed at def\textsubscript{4} and is overwritten at rd\textsubscript{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.

In order to derive feasible shared variable DU-paths on system level, we require (1) task-level information regarding when shared variables may be accessed by each task in the system, and (2) system-level information regarding how these tasks are scheduled and temporally interfere.

2.2 Contribution

The method presented in this paper fulfils these requirements, and derives 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.

2.3 System Model

We assume a uni-processor single node 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. The lowest priority (0) is assigned to the idle task, which executes when no other tasks are scheduled. We assume that the programming language is of an imperative type, and that task code is written without recursive constructs and with bounded loops.

Traditionally, a real-time task is represented as a 4-tuple, $<T, O, P, D>$, where $T$ is the periodicity of the task. The task’s release time for each period is calculated by adding the offset $O$ to $T$. For all released tasks, the scheduling mechanism determines which task that will execute based on the task’s priority, $P$. In the task’s period $T$, latest completion time is determined by the task’s deadline, $D$. In this paper, we extend a task to 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 ($D$) and uses ($U$) of the data. Hence, we represent a task as a 7-tuple, $<T, O, P, D, ET, D, U>$. Note that the $T, O, P, D$ properties are given by the system designer, whereas properties $ET, D, U$ are derived by the analysis in stage 1 (see Section 3.1).

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 be spawned into one or more jobs. A job inherits the $P$ and $D$ properties of its native task, and its release time $R$ and deadline are calculated using the task $T$, $O$, and $D$ properties respectively.

3 System-Level DU Analysis

In this section, we describe how to derive task-level information regarding when shared variables may be accessed by each task in the system (Section 3.1), and how 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 $v \in W_S$. 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 execution time from the start of the task to the statement containing $d$. The $d.\max$ property describes the longest possible execution time from the start of the task to the statement containing $d$. The $d.rdMax$ property describes the longest possible execution 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 undoubtedly overwritten.

3.1.1. Timing analysis for defs and uses

The SWEET tool (SWEdish Execution time Tool) [5] is a research prototype tool developed at Mälardalen University [10]. SWEET consists of three distinguished phases: a flow analysis where bounds on the number of different entities in the code can be executed is derived, a low-level analysis where bounds of the execution times for instructions are derived, 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 producing estimates upon the above mentioned $\min$, $\max$, and $rdMax$ values.

We initially 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) contain some example code containing two globals $g$ and $p$. Figure 4(b) illustrates the CFG for the code in Figure 4(a). The flow analysis has derived a loop bound of 10, expressed as an upper bound on the number of times BB4 could be executed. Each node has also been given a timing by the low-level analysis, valid each time the node is taken.

Secondly, we perform a reaching definition (RD) analysis for the global variables in the program [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 preceding 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/entities} x_i * t_i, subject to constraints reflecting the structure of the program and possible flows. For example, there are constraints specifying that the start and exit nodes each must be taken exactly once, and constraints specifying 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 derived by minimizing the same sum, subject to the same constraints.

All our analyses start from the above mentioned graph. Depending on what timing values that should be derived we modify the graph somewhat by adding extra edges and flow constraints. For example, the graph for deriving min, max for a use \( u \) is constructed by adding en extra edge from the basic block 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-edges and flow contraints. For example, the latter denoting the number of times that block or edge is executed. The WCET is found by maximising the sum \sum_{i/entities} x_i * t_i, subject to constraints reflecting the structure of the program and possible flows. For example, there are constraints specifying that the start and exit nodes each must be taken exactly once, and constraints specifying 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 derived by minimizing the same sum, subject to the same constraints.

All our analyses start from the above mentioned graph. Depending on what timing values that should be derived we modify the graph somewhat by adding extra edges and flow constraints. For example, the graph for deriving min, max for a use \( u \) is constructed by adding en extra edge from the basic block 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-edges and flow contraints. For example, the latter denoting the number of times that block or edge is executed. The WCET is found by maximising the sum \sum_{i/entities} x_i * t_i, subject to constraints reflecting the structure of the program and possible flows. For example, there are constraints specifying that the start and exit nodes each must be taken exactly once, and constraints specifying 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 derived by minimizing the same sum, subject to the same constraints.

All our analyses start from the above mentioned graph. Depending on what timing values that should be derived we modify the graph somewhat by adding extra edges and flow constraints. For example, the graph for deriving min, max for a use \( u \) is constructed by adding en extra edge from the basic block 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-edges and flow contraints. For example, the latter denoting the number of times that block or edge is executed. The WCET is found by maximising the sum \sum_{i/entities} x_i * t_i, subject to constraints reflecting the structure of the program and possible flows. For example, there are constraints specifying that the start and exit nodes each must be taken exactly once, and constraints specifying 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 derived by minimizing the same sum, subject to the same constraints.
Definition rules:
1. At \( d.\min \), \( d \) makes a transition from \( \text{dead} \rightarrow \text{active} \).
2. At \( d.\max \), \( d \) makes a transition from \( \text{active} \rightarrow \text{live} \).
3. At \( d.rdMax \), \( d \) makes a transition from \( \text{live} \rightarrow \text{dead} \).

Use rules:
1. At \( u.\min \), \( u \) makes a transition from \( \text{dead} \rightarrow \text{active} \).
2. At \( u.\max \), \( u \) makes a transition from \( \text{active} \rightarrow \text{dead} \).

DU-path rules:
1. At \( d.\min \), all DU-paths \((d, u)\), such that \( u.var = d.var \) and \( u \) is currently active, are added to the analysis result.
2. At \( u.\min \), all DU-paths \((d, u)\), such that \( u.var = d.var \) and \( d \) is currently live or currently active, are added to the analysis result.

These seven rules (partially depicted in Figure 5) are implemented in the DUANalysis algorithm, deriving all system-level DU-paths.

3.2.1. The DuAnalysis Algorithm

The DUANalysis algorithm (Figure 6) 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 \( \text{State} \) 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. \( \text{State} \) represents the current abstract state of the execution, and contains information regarding currently live definitions, active definitions, active uses, and encountered DU-paths. Hence, \( \text{State} \) is defined as:

\[
\text{State} : \{ \text{liveDefs}, \text{activeDefs}, \text{activeUses}, \text{duPaths} \}
\]

Transition represents the change of state incurred by the execution of a certain part of a job. As such, Transition contains new active definitions, killed live definitions, killed active definitions, killed active uses, and the WCET of the executed job. Hence, Transition is defined as:

\[
\text{Transition} : \{ \text{newActiveDefs}, \text{killedLiveDefs}, \text{killedActiveDefs}, \text{killedActiveUses}, \text{wcet} \}
\]

Intuitively, the combination of a \( \text{State} \) \( a_1 \) and a Transition \( a'_1 \) yields a new \( \text{State} \) \( a_2 \), 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

\[
\text{EXECUTE} : \text{State} \times \text{Job} \times \text{Ivl} \times \text{Ivl} \rightarrow \text{Transition}
\]

\[
\text{SWITCHTASK} : \text{Transition} \times \text{State} \times \text{Ivl} \rightarrow \text{State}
\]

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 described above.

Two more structures (Ivl and Job) are used in the analysis. Ivl defines a time interval by its extremal values \( f \) and \( r \). Job represents a task instance and contains definitions, uses, job priority, release time, BCET and WCET. Job is defined as:

\[
\text{Job} : \{ D, U, P, R, BCET, WCET \}
\]

Roughly, the DUANalysis algorithm starts with an empty \( \text{State} \) 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 \( \text{State} \) to a new \( \text{State} \). 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 \( \text{State} \) to a new \( \text{State} \) - 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 and its auxiliary
functions, ExecDef and ExecUse (Figure 7) works through all shared variable accesses in \( j \) in a chronological order. Each access is treated according to its corresponding definition or use rule (definition rules are implemented in ExecDef and use rules are implemented in ExecUse).

SwitchTask and its auxiliary functions ReviseDef and ReviseUse (Figure 8) create 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. Add new DU-paths to the result of the analysis SwitchTask implements DU-path rule 1 in ReviseDef, and DU-path rule 2 in ReviseUse.

Note that, compared to the original EOG algorithm, DuANALYSIS uses a number of additional auxiliary functions (for a description of all auxiliary functions, ExecDef and ExecUse (Figure 7) works through all shared variable accesses in \( j \) in a chronological order. Each access is treated according to its corresponding definition or use rule (definition rules are implemented in ExecDef and use rules are implemented in ExecUse).

SwitchTask and its auxiliary functions ReviseDef and ReviseUse (Figure 8) create 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. Add new DU-paths to the result of the analysis SwitchTask implements DU-path rule 1 in ReviseDef, and DU-path rule 2 in ReviseUse.

Figure 6. The DuANALYSIS algorithm.
\textbf{Table 1. Evaluation Results.}

<table>
<thead>
<tr>
<th>System</th>
<th>Tasks</th>
<th>GVar</th>
<th>$C_{DU}$</th>
<th>$F_{DU}$</th>
<th>$F_{DU}$ / $C_{DU}$</th>
<th>Config1</th>
<th>$F_{DU}$</th>
<th>$F_{DU}$ / $C_{DU}$</th>
<th>Config2</th>
<th>$F_{DU}$</th>
<th>$F_{DU}$ / $C_{DU}$</th>
<th>Config3</th>
<th>$F_{DU}$</th>
<th>$F_{DU}$ / $C_{DU}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>4</td>
<td>18</td>
<td>216</td>
<td>155</td>
<td>71.6%</td>
<td>147</td>
<td>68.1%</td>
<td>134</td>
<td>62.6%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td>4</td>
<td>27</td>
<td>183</td>
<td>N/A</td>
<td>N/A</td>
<td>176</td>
<td>96.2%</td>
<td>163</td>
<td>89.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td>4</td>
<td>22</td>
<td>34</td>
<td>32</td>
<td>94.1%</td>
<td>32</td>
<td>94.1%</td>
<td>27</td>
<td>79.4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_4$</td>
<td>3</td>
<td>7</td>
<td>44</td>
<td>34</td>
<td>77.3%</td>
<td>37</td>
<td>84.1%</td>
<td>24</td>
<td>54.5%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_5$</td>
<td>2</td>
<td>4</td>
<td>236</td>
<td>204</td>
<td>86.4%</td>
<td>196</td>
<td>83.1%</td>
<td>180</td>
<td>76.3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_3$), each scheduled in three different ways (Config1-3). All systems comprise control-oriented code (e.g., calculation of planet orbits ($S_1$) and a control system for a forklift able to solve the Towers Of Hanoi problem ($S_2$)). In Table 1, Tasks 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 form a DU-path $(d,u)$). $F_{DU}$ refers to the number of DU-paths found feasible by the DUANALYSIS algorithm. Hence, $F_{DU} / C_{DU}$ describes the ratio between the number of DU-paths found feasible by our algorithm, and the combinatorially feasible DU-paths. As for the configurations, in Config3, the tasks are completely separated in time and suffer no preemptions. In contrast, Config1 is scheduled to maximize the number of task preemptions. Config2 is an in-between configuration of Config1 and Config3. Generally, with a less complex scheduling, more DU-paths are found feasible (except for Config1 and Config2 of $S_4$). Note also that the system-level analysis of Config1 of $S_4$ proved too complex to execute. Since all other configurations finished within a few seconds, we will investigate this problem further.

\section{Conclusion and Future Work}

Unit level testing can reveal many failures early in the testing phase and can make a huge impact on the software quality. However, 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, but we have found no evidence that our method could not be used on a more relaxed system model, e.g., a system model based on transactions of task instead of strictly periodic tasks. Our method however requires a finite (periodically or non-periodically) repeated system behaviour.

Testing at system level is often performed after either the specification or the implementation has been changed. Here, it is important to establish if the correction had the desired effect and that the failure is removed. This type of testing is known as \textit{confirmation} testing. We intend to extend our test method and analysis in order to support confirmation testing of multi-tasking real-time systems. Furthermore, we intend to extend our test method and analysis in order to support regression testing (i.e., the testing performed to establish that the changes do not introduce any new faults).

\section{References}


```c
void reviseDef (t, transition, state)
{
    for each u ∈ state.activeUses // For all currently live defs
        if (t < d.min) ∧ (d.var = u.var)
            // New DU-path found. Add it!
            state.activeDefs = state.activeDefs \ {d}
    for each d ∈ transition.killedActiveDefs // For killed active defs.
        if t < d.max
            // Remove d from the set of active defs.
            state.activeDefs = state.activeDefs \ {d}
    for each u ∈ transition.killedLiveDefs // For killed live uses.
        if t < u.max
            // Remove u from the set of live uses.
            state.liveDefs = state.liveDefs \ {u}
    return state
}

void reviseUse (t, transition, state)
{
    for each d ∈ state.liveDefs // For all currently live defs
        for each u ∈ transition.nonActiveUses // and new live uses.
            if (t < u.min) ∧ (d.var = u.var)
                // New DU-path found. Add it!
                state.duPaths = state.duPaths ∪ \{(d, u)}
    for each u ∈ transition.killedLiveDefs // For killed live uses.
        if t < u.max
            // Remove u from the set of live uses.
            state.liveDefs = state.liveDefs \ {u}
    return state
}

switchTask (state, transition, TSI)
{
    t = getNextDUChange(0, transition)
    while t < TSI.r
        state = reviseDef(t, transition, state)
        state = reviseUse(t, transition, state)
        t = getNextDUChange(t, transition)
    if (TSI.t ≠ TSI.r) ∧ (TSI.r < transition.WCET)
        // A non-preempted non-wcet branch.
        for each d ∈ transition.killedActiveDefs
            if d.max > TSI.r
                // Remove d from the set of active defs.
                state.activeDefs = state.activeDefs \ {d}
        for each d ∈ transition.killedLiveDefs
            if transition.WCET > d.max > TSI.r
                // Remove d from the set of live defs.
                state.liveDefs = state.liveDefs \ {d}
    return state
}
```

Figure 8. The `switchTask` functions.