Relaxing Event Densities by Exploiting Infeasible Paths in Control Flow Graphs

Kilian Kempf, Steffen Kollmann, Victor Pollex, Frank Slomka*
Institute of Embedded Systems/Real-Time Systems
Ulm University
{firstname}.{lastname}@uni-ulm.de

Abstract

Common real-time analysis techniques for embedded systems mainly concentrate on a task model where every single activation of a task leads to a single outgoing event that is emitted at the end of the task's computation. An extension has been introduced that allows for multiple events to occur during a single job by respecting the control flow inside a task in order to recalculate the worst-case density for outgoing events. This previous work does however not consider the effect that data dependencies have on the calculation, which leads to pessimistic results caused by infeasible paths inherent in the control flow. In this paper we propose a method involving a flow graph transformation that reduces this pessimism when using the existing analysis and show its integration into a reasonable workflow.

1 Introduction

The process of developing embedded systems involves a variety of steps. One of them is the mapping of software components onto the hardware resources of the system in development. An integral part of this mapping process is the analysis of the timing behavior of the multitude of included software components. This is of great importance in order to verify the compliance with the time constraints of the systems imposed by the context.

The various software components and functions of an embedded system are usually modeled and implemented as software tasks. They are activated by external events originating from the system context and are able to generate events themselves in order to activate each other. This leads to the concept of a task chain, where several tasks are depending on events from their predecessors. In order to analyze such a scenario, the general approach is to determine the maximal (and minimal) amount of time that would be needed to complete the task if it had the entire resource it is to be mapped on for itself. According to common models, an event is generated when the task is completed. This event is then used to activate other dependent tasks or to interact with the system's context.

Different types of real-time analyses now consider the fact that current systems have multiple tasks mapped onto the same resource and perform preemptive multitasking. There are many possibilities of scheduling that all lead to the time-sharing of pseudo-concurrent tasks that are frequently interrupting each other. The real-time analysis tries to consider the high dynamics inherent in such systems and is able to give guarantees for the timed execution of the tasks which generate the next event in the task chain. The analysis of a complete system determines the time requirements for all tasks involved. These time requirements in turn directly dictate the requirements on the resources. Therefore, minimizing the pessimism of the analysis also minimizes the necessary resources in terms of space, energy and ultimately costs.

Still, the common analysis models and techniques regard a task as a unit that may be interrupted and therefore delayed, but which starts with the consumption and ends with the generation of an event. A technique called event dependency analysis has been developed by Bodmann et al. [5] which considers the internal structure of a task and extends the common model by the possibility to generate an event before the task has finished its execution. This is closer to reality and allows to relax the estimated temporal density of the resulting events which in turn enables to make less pessimistic assumptions. The technique bases on the control flow graph which consists of basic blocks and already is fundamental for the worst-case execution-time (WCET) analysis, a method that estimates bounds for the execution time of a task.

The graph is extended by blocks that generate events so that for every number of events, the minimal time interval in which they may occur can be extracted. After taking steps to integrate the effects of recurring jobs it allows to calculate for a given spectrum of incoming events an associated spectrum of outgoing events that is usually less pessimistic than the ones calculated by the common models. This is well desired because as explained above less pessimism leads to a reduction of resource requirements and therefore costs.

*This work was supported by the German Research Foundation.

The aim of this paper is to extend the task-based analysis method mentioned above and to integrate the consideration of data dependencies inherent in the tasks under analysis. While the existing analysis considers all possible paths in the control flow of a task, we now attempt to determine the relevant ones more precisely. This allows to restrict the pessimism by eliminating paths that are not feasible.

We propose a transformation of the flow graphs that allows for a simplified way of eliminating those infeasible paths so that the existing analysis can be kept. The identification of the paths itself is handled by a WCET analysis tool, as these tools generally need to detect infeasible paths in order to tighten their estimations. The determined flow facts (also called flow constraints) describe dependencies inside the control flow.

Our approach consists of three steps. First, we split up execution paths in the flow graph. The flow graph is then processed by a WCET analyzer which detects the infeasible paths. Afterwards we are able to eliminate the paths by pruning the graph. The modified graphs then are free of the inherent pessimism. This leads to a relaxation of the event densities which in turn results in a more accurate analysis. Once again, helping the analysis to determine better estimates of the timing requirements causes less overestimation of the required resources.

This paper is organized as follows: after giving an overview on the related work in Section 2, the model we use for the system level and the task level is described in Section 3. The general workflow presented in Section 4 gives an overview of our approach. Section 5 introduces the current state of the event dependency analysis we aim to improve. Afterwards, our proposed transformation is presented in detail (Section 6). The benefit of the approach is then demonstrated by a small example in Section 7 before the paper ends with a short conclusion.

2 Related work

Analyzing distributed real-time systems is still a challenge for scientists. The problem is not the calculation of guaranteed bounds for the worst-case timing behavior of tasks in a system but the calculation bounds that are tight. For simple systems the approach of Tindell and Clark [24] delivers good results, but if the complexity of the system increases the bounds get worse. The consequence was that more sophisticated approaches like the SymTA/S approach [17] or the Real-Time Calculus [6] have been developed. These approaches are able to classify more systems as feasible. But even if the system is feasible, meaning that the deadlines in the system are not violated, the bounds have a direct impact on the design flow. In today's embedded systems, e.g. in the automotive or avionic industry, a common problem is the mapping of tasks to the processor units so that an optimal system architecture concerning costs, speed and power is designed. This results in tight bounds for the worst-case behavior being needed in the design flow so that an optimal system solution can be found. One problem leading to bad bounds is that during a real-time analysis the interference between the tasks is always maximal.

Many approaches have been developed in the past to improve the worst-case response time analysis by including task dependencies. For example, to consider offset dependencies between task stimulation the transaction model introduced in [23] and improved in [13] has been developed. To consider precedence correlations between tasks in a distributed system methods in [16] and [9] have been developed. Another idea is to consider the dependency caused by a non-preemptive scheduling as presented in [19]. The concept is that the tasks cannot be preempted and therefore the events for successive task must occur time-shifted. The common purpose of all these approaches is bounding the occurrence of given events.

Another layer which is orthogonal to the approaches mentioned above is the consideration of event types. Here the idea is that different types of events result in different execution times of the tasks. The the assumption that each job of a task is executed in its worst-case execution time is relaxed. A general model for this has been developed by Baruah in [3] which is based on [12] and has recently been extended in [21]. Approaches to consider the propagation of different event types through a system have been introduced in [18] by a hierarchical event model and in [14] by so-called event count curves. The result of these papers is that tighter end-to-end delays in a system can be calculated when the different event types are considered. The common idea of these approaches is to bound the interference of the assumed execution times.

The lack of all the techniques described above is that the task execution itself and the resulting events are not connected. The control flow of the task is not considered by the real-time analysis techniques although this is very important for bounding the events produced by a task. In order to overcome this issue we introduce a common approach considering data dependencies on the task level which can be connected to any real-time analysis method. We will cover a topic missing in the literature where we consider event dependencies caused by the control flow of a task.

3 Model

In this chapter we present the task model and the event model used in the paper. There are two different levels or views to be distinguished. One is the system level, where the different tasks, their activations and their interdependencies are examined. This is the level where real-time analysis is performed. The other view is the task level. Here, the focus is on the tasks themselves. This is the level where the worst-case execution time analysis usually takes place.

The following part describes the system level model we use. Generally speaking, all the tasks to be modeled for a specific system form the task set which consists of several individual tasks.

Definition 1 Task: *A task τ is a tuple $\tau = (c^+, c^-, d)$, where c^+ is the worst-case execution time, c^- the best-case execution time and d a relative deadline. Individual instances of a task are called jobs, where $\tau_{i,j}$ denotes the j-th job of task τ_i.*

Definition 2 Task set: *A task set Γ is the set that includes all tasks τ of a system.*

Inside a system, several tasks may be associated with each other. Their dependencies form a task graph where the vertices are the tasks while the directed edges describe the communication dependencies between the tasks.

Definition 3 Task graph: *A task graph is a directed graph $G = (\Gamma, E)$ with $E \subseteq \Gamma \times \Gamma$. $(\tau_1, \tau_2) \in E \Leftrightarrow \tau_1$ activates τ_2.*

In this paper we use a general and abstract event model. We define an event function η that describes for every time interval the number of events that can occur in this interval.

Definition 4 Event function: *The event function $\eta([s,t)) : \mathbb{R}^+ \mapsto \mathbb{N}^+$ denotes for any time interval $[s,t)$ the number of events that occur in that interval.*

Now we are able to extend the task graph to include the stimulations between the tasks. This is done by weighting the edges of the task graph to form an event dependency graph [5]:

Definition 5 Event dependency graph: *An Event dependency graph D is a task graph $D = (\Gamma, E)$ in which every edge is weighted with an event function η. $\eta_{1,2} := \tau_1$ stimulates τ_2 with η.*

After the definition of the model we use for the system level, we now present the model for the task level. The following definitions closely resemble the ones introduced by Allen in [2]. The tasks themselves may be modeled by a control flow graph, which is a directed possibly cyclic block-graph. The vertices are the blocks while the edges form a possible path for the flow of control.

Definition 6 Basic block: *A basic block (BB) is a sequence of instructions in a task that can be processed unconditionally, i.e. it contains no conditional jumps. The whole basic block can be the target of a jump. At the end of a basic block there may be a conditional jump to another basic block. All blocks are annotated with an execution time c which denotes the time needed for the uninterrupted execution of the block on a specific resource.*

Variation of those times caused by variable resources' properties depending on state or history like caches of processing elements are not considered here. As for this paper we are interested in the highest density of events, we use the best-case execution time of a basic block if its execution times may differ.

Definition 7 Control flow graph*: A control flow graph (CFG) is a directed graph $G = (V, E, s, x)$, where V is a set of basic blocks and $E \subseteq V \times V$ the set of edges between them that model the transfer of control. $s \in V$ denotes the single start node ($\forall v \in V : (v, s) \notin E$) and $x \in V$ denotes the single exit node ($\forall v \in V : (x, v) \notin E$). We demand that for every block $v \in V$ there are at most two edges to other blocks:*

$$\forall v \in V, O = \{v\} \times V : |E \cap O| \leq 2$$

Definition 8 Predecessors and successors: *For each node $v \in V$ of a CFG $G = (V, E, s, x)$ there is a set of predecessors pred(n) and successors succ(n):*

$$\begin{aligned} k \in \mathit{pred}(n) &\Leftrightarrow (k, n) \in E \\ k \in \mathit{succ}(n) &\Leftrightarrow (n, k) \in E \end{aligned}$$

Definition 9 Control blocks: *A basic block $v \in V$ of a CFG $G = (V, E, s, x)$ with $|succ(v)| = 2$ is called a control block. Every control block has an associated predicate expression A that can be evaluated to either TRUE or FALSE during the execution of the task. To indicate which successor will be taken for the control flow it also has the attributes* true *and* false *that are each set to a basic block $w \in succ(v)$ and indicate the successor to be taken in case expression A is evaluated to TRUE or FALSE respectively.*

The control flow during the execution of a task is modeled as a walk through the CFG of that task that starts at the start block and ends at the exit block.

Definition 10 Control flow: *The control flow through a CFG $G = (V, E, s, x)$ is a sequence S of nodes from the start block to the exit block*

$$S = (s, n_1, ..., n_n, x) \text{ with } s, x, n_{1..n} \in V.$$

Definition 11 Dominance: *Let $G = (V, E, s, x)$ be a CFG. A node $n \in V$ of that graph is dominated by a node $k \in V$ when every path from the start node s to n passes through k, that is if all of those paths are of the form $(s, ..., k, ..., n)$. We then write k dom n. Every node dominates itself: $\forall n \in V : n$ dom n.*

We now extend the model of a control flow graph for the generation of events during the execution of a task. As mentioned above, the common approach is to model all events that are generated during the execution of a task as if they occurred at the very end of the task's execution. The introduction of event blocks allows us to model the events at the times they actually occur. Event blocks are basic blocks that generate an event at the end of their execution. The event blocks trigger task-global events, which means that all subsequent dependent tasks are activated by the event.

Definition 12 Event block: *An event block n is a basic block inside a task τ_i for which event$(n) = $ true, meaning that it generates an event which activates another task τ_j.*

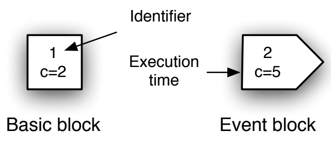

Figure 1. Graphical representation of basic block and event block

The graphical representation for an event block we use in this paper is a block arrow that points to the right and resembles the output symbol of the Specification and Description Language (SDL). Figure 1 depicts such an event block along a normal basic block.

Definition 13 Event flow graph: *An event flow graph (EFG) is a modified control flow graph* $G = (V, E, s, x)$ *with* $\exists n \in V : event(n) = true$.

We are now able to model a system with tasks and the stimuli between them as well as the tasks themselves and the positions inside the structure of the tasks that generate events activating other tasks.

4 Workflow

As previously mentioned, we harness the power of an established worst-case execution time analysis tool as we focus on the events in and between tasks. The following workflow therefore contains a tight coupling between our approach and a WCET tool. An important element are the flow facts that are generated during the WCET analysis. Analyzers using a technique called implicit path enumeration (IPET) [11] determine flow facts that describe dependencies in the control flow of a task. We use these flow facts as an interface to the WCET tool. An overview of currently used approaches for WCET analysis has been presented in [15] and [25]. Our idea is to provide a pre-processed flow graph for the analysis that facilitates the post-processing which removes the detected infeasible paths before performing the event dependency analysis [5] as described in section 5.

The anticipated workflow is depicted in Figure 2. It starts with the task graph, which identifies all relevant tasks and their relationship to each other and can either be extracted automatically or be generated by hand. We believe that the manual approach might be more common as it allows for greater flexibility. In that case, the analysis of a system starts with the modeling of the tasks in the analysis tool after which their program code is imported. All aspects of this first step are outside the scope of this paper. The next steps are the ones relevant for our approach and they have to be performed for every task that is going to be analyzed.

First the basis blocks are extracted and the control flow graph is constructed. This may be the work of a compiler or a dedicated analysis tool. Program code in the form

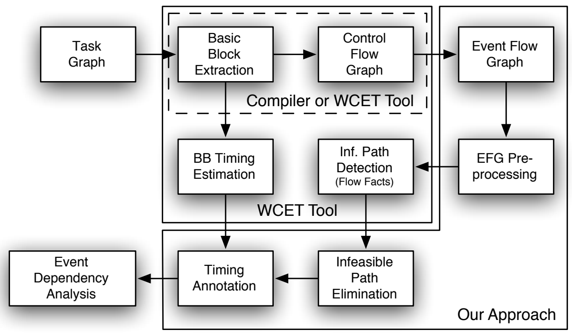

Figure 2. Suggested workflow

of a high(er) level language (i.e. C code) might better be treated by a compiler framework while machine code has to be processed by a specialized tool. This might be done by a WCET analyzer.

The control flow graph then has to be extended into the event flow graph. A mapping of all program calls that generate events for other tasks has to be supplied. If the task graph has been extracted automatically, the necessary information should already be present as it was already needed to determine the tasks' interdependencies. If more than one task is activated, the event flow graph needs additional information for the grouping of event blocks.

Afterwards, the resulting event flow graph is pre-processed. This involves mainly the transformation introduced in Section 6.1. The resulting modified event flow graph contains a significant number of split-up paths that are connected to a single exit node.

The modified flow graph is then processed by a WCET analyzer. In this step, the detection of the infeasible paths (see Section 6.2) takes place. What we are really interested in is the annotation of the flow graph with flow facts, from which we are able to infer the infeasible paths.

With the help of the flow facts we may then eliminate the infeasible paths of the event flow graph. This is possible because we had already modified it in a way that every infeasible path leads into a separate subtree that may be deleted from the event flow graph without affecting other paths. Section 6.3 shows how.

Independently we use the WCET analysis tool to estimate the timing for every basic block. This is done for the earlier extracted set of basic blocks that formed the initial control flow graph, because the modified event flow graph we obtained in the last step contains many duplicated blocks that would unnecessarily slow down the WCET tool. As we eventually are interested in the greatest possible density of events, we need the best case execution time of the basic blocks. The result of this step is an annotated control flow graph.

In the final step we annotate the modified event flow graph with the estimated execution times. The necessary mapping between the already annotated control flow graph and the modified event flow graph is possible because our workflow did not alter the basic blocks themselves. The resulting flow graph can then be processed by the regular event dependency analysis as introduced by Bodmann et al. [5] which is described in the following section.

5 Event dependency analysis

We already mentioned in the introduction as well as in the section on the related work that the general approaches used in real-time analysis do not consider the effects that the internal structure of tasks has on the density of events. The models described in Section 2 usually assume that only one event is generated during the execution of a job and this event is emitted when the job finishes its computation.

The extension of this perception to allow for multiple events to emerge during a single job is not trivial. Clearly it is not acceptable to adhere to the established notion and accumulate the events until the end. As this would result in multiple events being emitted at the same time, the inherent pessimism will very likely consider many systems infeasible that in reality would work well. The solution can only be to enforce a certain minimal distance between the events produced by a job, as in reality they will not occur at the same time.

As noted before, a reasonable way to determine the minimal distance between the events has to take a task's internal structure into consideration. The control flow graph of a task offers a sensible way to do that. If the positions inside a task where the events are generated as well as the possible flows of control are known, the time between the occurrence of events during the execution may be concluded.

Bodmann et al. took this approach in [5]. They base their work on a control flow graph that has been extended by the possibility to describe the appearance of events within the flow. This is the event flow graph. When the execution times of all basic blocks in the event flow graph are known, it is possible to determine the minimal amount of time that passes between events during the execution of the task.

Advanced models for real-time analysis like [6] demand that not only the minimal distance between two events but also that between three, four, five, etc. is known. Bodmann et al. cover this condition by defining a set of functions which denote for a given number of events the minimal time interval in which they may occur. Hence these functions are called interval functions.

The function *inI* does this for intervals that may reside anywhere inside the event flow graph as long as they occur during the execution of a valid control flow. If the number of desired events is greater than the one which the task is able to generate, inI becomes infinite. There are two additional interval functions *startI* and *endI*. For them the intervals are bound to the start and the end of the task, respectively. These two functions are used for the formal definition of inI and additionally become necessary when

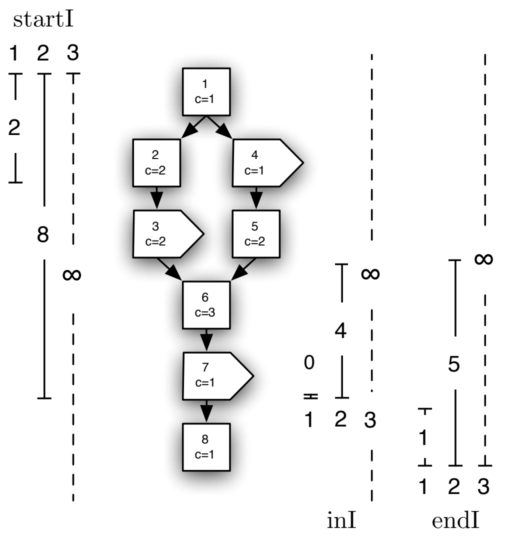

Figure 3. Elementary interval functions in an event flow graph

the activation of the examined task with multiple external events is considered. Again, for an infeasible number of events the interval functions become infinite.

Example 1 *An example of an event flow graph where the interval functions are annotated is depicted in Figure 3. Note that* $inI(1) = 0$ *because in this case the time interval in which one event may occur is infinitesimal. The interval could have also been annotated at the end of any other event block.*

In [5] Bodmann et al. have also presented an algorithm allowing for an efficient calculation of the interval functions. It traverses an event flow graph from top to bottom while visiting any node exactly once. Bodmann et al. have presented definitions of the interval functions that base on the elementary graph operations *concatenate* and *merge*. All flow graphs may be constructed and deconstructed with these two operations. Accordingly, the analysis algorithm is able to iteratively determine the interval functions for any node it encounters during the traversal of a graph.

For a single activation of a task the density of outgoing events is denoted by the inI function. In contrast, when multiple activations of the task are taken into consideration the time intervals may be compressed. If the execution of a job is delayed, its end time may be close to the next activation of the task and therefore to the starting time of the next job. In that case the minimal time interval in which a given number of events occur may stretch across both jobs.

This is also generally the case if the desired number of events is greater than the one a single job is able to generate. The thought may be extended to an arbitrary number of events. Any time interval that stretches over multiple jobs may be arranged in a way that it reaches into the end of the first job and into the start of the last job. At this point

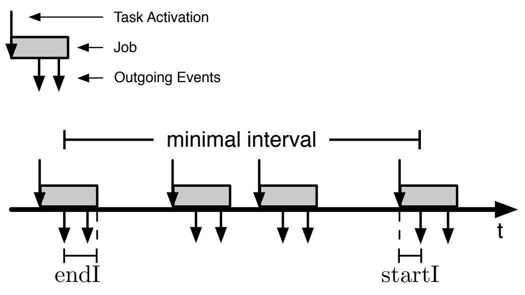

Figure 4. Handling of multiple external events

the interval functions startI and endI are applied. For the inner jobs that are fully enclosed by the interval the maximal number of events they can generate is assumed. The remaining events have to be divided between the first and the last job using the functions endI and startI respectively while minimizing the overall interval. Figure 4 shows an example with an interval for seven outgoing events that stretches across four jobs.

The event dependency analysis as presented in [5] requires loops in the control flow of tasks to be unrolled before they are processed. Albers et al. have introduced an extended event model in [1] that facilitates the handling of loops for the event dependency analysis.

6 Considering infeasible paths

The previous section presented the current state of the event dependency analysis. This section introduces an extension to that technique which results in a relaxation of the event streams.

The current event dependency analysis assumes the smallest interval between events when merging branches which leads to the highest possible density of events. This introduces unnecessary pessimism when some combination of branches may never be executed together during a single run of the task, which means that infeasible paths exist in the event flow graph. Our approach is to modify the graph in a way that eliminates the infeasible paths so that it may be used as an input to the existing event dependency analysis. The reduction of pessimism leads to an relaxation of the outgoing event density which in turn will reduce the processing power or timing requirements necessary to guarantee a reliable system.

Example 2 *An example of infeasible paths is illustrated in Figure 5. Given that $A \Leftrightarrow B$ and the value of the corresponding variables are not changed in one of the subsequent blocks, only two events will be generated during the execution of the pictured program stub. The significant interval functions startI, endI, inI will also differ if the the infeasible paths are respected.*

Although this might be regarded as overly simplified, the implications are quite relevant. According to Stein and

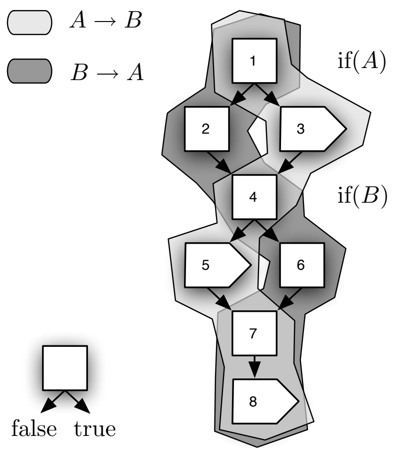

Figure 5. An example of infeasible paths

Martin [20] the existence of such constructs in embedded systems code is not uncommon as they can be the result of automatically generated code. Suhendra et al. [22] refer to code generated from a state chart that contains a lot of repetitive checks leading to many infeasible paths.

We call this type of infeasible paths mutual exclusion in the control flow of tasks. The term mutual exclusion is used here to denote that the same condition that directs the control flow into a certain direction is re-evaluated at a subsequent control block in the flow graph while the variables relevant for a control flow decision are not adversely affected by the intermediary blocks. This leads to a reduction of the feasible combinations of subpaths, therefore eliminating certain otherwise possible paths through the graph. The subpaths mutually exclude each other.

Definition 14 Infeasible Path:
A path $P = (n_1, ..., n_n, c, b, m_1, ..., m_m)$ is said to be infeasible if c is a control block containing the predicate expression A which always evaluates to the same logical value when reached through the path $P' = (n_1, ..., n_n, c)$ while $b \in succ(c)$ is the node following the control block that would be taken if A evaluated to the opposite value.

Inspired by the definition of an infeasible path that is irreducible given by Bodík et al. in [4] we adopt:

Definition 15 Shortest Infeasible Path: *An infeasible path $P = (n_1, n_2..., n_n)$ is a shortest infeasible path if both subpaths $(n_2, ..., n_n)$ and $(n_1, ..., n_{n-1})$ are feasible.*

We will now present our approach for the handling of control flow graphs that make it possible to exploit inherent infeasible paths. Our idea is to transform the graphs in a way that makes them usable for the existing analysis presented in Section 5. The objective of this transformation is to obtain graphs that allow an elimination of infeasible paths through pruning. We concentrate on the events in and between tasks, therefore we interface with a worst-case execution time analyzer. Those tools include the identification of infeasible execution paths in general as an inherent feature. This is what we take advantage of.

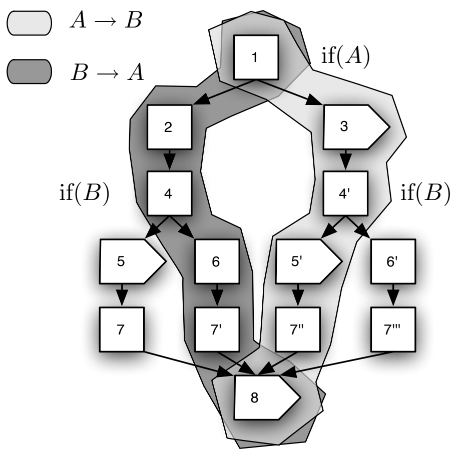

Figure 6. Infeasible paths in the modified event flow graph

The transformation we propose basically splits all possible execution paths in the control flow graph. If the graph contained infeasible paths this will result in the existence of infeasible subgraphs. We call this the pre-processing of the event flow graph. In a second step, these subgraphs have to be identified, which is done with a WCET analyzer. After their identification, the infeasible subgraphs can be completely eliminated from the control flow graph. This is the post-processing. The resulting reduced graph is free of any detectable inherent static mutual exclusion and can be handled with the exisiting event dependency analysis.

6.1 Pre-processing the event flow graph

Our transformation traverses the complete event flow graph (the source) and constructs a new event flow graph (the target) by copying basic blocks to the new graph. For every control block that is encountered while traversing the source, both possible paths are followed separately.

The source graph may contain loops that may or may not originate from back edges [10]:

Definition 16 Back edge:
A back edge is an edge whose head dominates its tail: $\forall (k, n) \in E : (k, n)$ *is a back edge* $\Leftrightarrow n$ *dom* k.

Back edges usually originate from higher-level control structures like *while* and *for*, whereas the use of *goto* (or generally jumps in assembler or machine code) may create loops without back edges. Graphs containing such loops are called irreducible graphs and may pose a significant challenge for the program analysis. Otherwise the graph is reducible [10]:

Definition 17 Reducible flow graph: *A flow graph* $G = (V, E, s, x)$ *is called reducible if and only if its subgraph* $G' = (V, E', s, x)$ *with* $E' = E \setminus \{(k, n) \in E | (k, n)$ *is back edge*$\}$ *is acyclic and every node* $v \in V$ *can be reached from the start node* s.

Various methods for the handling of irreducible control flow graphs have been developed. Janssen and Corporaal present one in [10] and reference several others.

Loops in the source graph are handled the following way: A loop is detected whenever the successor of the current node is already in the currently traversed path: $P = (n_1, ..., n_i, ..., n_n, c, n_i)$. Note that (c, n_i) is not necessarily a back edge.

A loop that does not contain any events itself ($\forall n \in \{c, n_i, ..., n_n\} : \neg event(n)$) cannot add events if taken repeatedly but may only stretch the time between events. The edge (c, n_i) is therefore not copied to the target. This resembles the worst case. Another possibility would be to keep the edge if a minimal number of loops is known. This could be used to further relax the event density.

A loop that contains at least one event ($\exists n \in \{c, n_i, ..., n_n\} : event(n)$) has to be preserved. An edge resembling (c, n_i) is therefore added to the target graph. This new edge is always a back edge. That way, the modified event flow graph is always reducible and the remaining loops may then be handled by the event dependency analysis.

The transformation creates a lot of leaves that are however all copies of the same basic block namely the former single exit node. Therefore at the end of the transformation all leaves are merged into a single node that is the new (and old) exit node. The resulting modified event flow graph for the example in Figure 5 is shown in Figure 6. A representation of the transformation in pseudo code is given in Listing 1.

Listing 1. Pseudocode of transformation

```
copy start node to target
push ({},start) to stack
WHILE not stack empty
  pop (path, current) from stack
  add current to path
  IF succ(current) == 2
    IF succ.false already in path
      link node.false to corresponding node
    ELSE
      copy succ.false to target
    END IF
    IF succ.true already in path
      link node.true to corresponding node
    ELSE
      copy succ.true to target
    END IF
    push (path, succ.true) to stack
    push (path, succ.false) to stack
  ELSE IF succ(current) == 1
    IF succ(current) already in path
      link to corresponding node in target
    ELSE
      copy succ to target
      push (path, succ) to stack
    END IF
  END IF
END WHILE
merge all nodes with (succ() == 0)
```

6.2 Detecting infeasible paths

We distinguish between two different kinds of detection possibilities for infeasible paths. The first one is the more intuitive one and is based on value assignments to variables. If in some node of a path there is an assignment that influences the condition of a control block following in that path in a way that completely evaluates its predicate expression, one of the two nodes succeeding the control block will never be reached and is therefore part of an infeasible path. The assignments can either be immediate by assigning a constant to the variable, or they can be the value of another variable which itself has been assigned a constant value that can be determined by a static analysis. Of course, along the path there may be a chain of assignments between variables that lead to the final outcome.

Let $P = (n_1, ..., n_n, c, b)$ be a path with a control block c and $b \in \mathrm{succ}(c)$ the first node in one of the branches of c. Let $A(a_1, ..., a_n)$ be the predicate expression which c evaluates. If the basic blocks on the path $(n_1, ..., n_n)$ statically set the variables $a_1, ..., a_n$ into a state so that node b is never taken, P is an infeasible path.

The second possibility to detect infeasible paths may be applicable when the value of an expression cannot be determined by a static value analysis. If the corresponding control block is preceded in the path by one whose expression is evaluated beforehand this may allow to deduce the value of the expression at hand. If at least one of the two expressions (or their negation) implies the other one (or their negation), one of the successors of the second may not be reached in that path as long as the variables of condition are not adversely affected in the basic blocks in between.

Let $P = (c_1, n_1, ..., n_n, c_2, f)$ be a path with control blocks c_1 and c_2 and $f \in \mathrm{succ}(c_2)$. Let A be the predicate expression that c_1 evaluates and B the one that c_2 evaluates. If $A \rightarrow B$ or $A \rightarrow \neg B$ or $\neg A \rightarrow B$ or $\neg A \rightarrow \neg B$, then P may be an infeasible path.

Let $\{a_1, ..., a_n\}$ be a set of variables on which A and B depend. Let us assume that $A \rightarrow B$ and we examine the path $Q = (c_1, t, ..., c_2, f)$ where $c_1.true = t$ and $c_2.false = f$. If none of $a_1, ..., a_n$ are altered in the path between c_1 and c_2, Q is infeasible because if expression A evaluated positively, expression B will also evaluate positively and the branch beginning with f is never taken.

But even if some of the variables are altered, an infeasible path may still exist. If the variables are redefined in a way that they satisfy a new expression A' for which it can be shown that $A' \rightarrow A$ it can be deduced $A' \rightarrow B$, which again makes Q an infeasible path.

Example 3 *Let $A = x > 2$ and $B = x > 0$ and path Q as above. Obviously $A \rightarrow B$. If node t redefined $x := x+1$ we gain a new expression $A' = x > 3$ with $A' \rightarrow A$. As we can deduce $A' \rightarrow B$, the path Q is still infeasible.*

Approaches for the detection and handling of infeasible paths have already been integrated into WECT analysis tools. In [20] Stein and Martin of AbsInt, the company behind the aiT WCET analyzer, presented their approach to the detection and elimination of infeasible execution paths. They show how program code on machine level can be analyzed, which they describe as more challenging than the analysis of high-level language code because of the added difficulty of determining the correct branching conditions. Stein and Martin propose a flow constraint analysis that aims at gathering the flow facts which are then solved using a theorem-prover framework. Another approach has been described by Gustafsson et al., who work on the SWEET WCET analyzer, in [7] and [8]. They introduce abstract execution, a variant of symbolic execution in order to determine infeasible paths. Three different algorithms are provided which result in the calculation of infeasible nodes, infeasible pairs and infeasible paths.

6.3 Eliminating infeasible paths

After the modified event flow graph has been processed by a worst-case execution time analysis tool, the infeasible paths should have been identified and annotated in the form of flow facts. This should include the ones found by a value analysis as well as those found by a comparison of the branching expressions.

An additional post-processing step is necessary to prepare the event flow graph for the event dependency analysis. In this step the infeasible paths are removed from the graph. Depending on the specific tool used for the analysis, the steps necessary for the actual extraction of the infeasible paths might differ a bit.

We assume that every branch that may never be taken is marked as infeasible. This may especially be the case if the corresponding infeasible path has been identified by a static value analysis. If the infeasibility has been detected by comparing branching conditions we might have to identify the branch by comparing the flow facts annotated by the analysis tool. In any case, the first basic block that is unreachable will be known. This is exactly the last node of a shortest infeasible path as given in Definition 15.

We will now describe the elimination of the infeasible paths. For every shortest infeasible path, the last node (the first node that is unreachable) is identified and removed from the modified event flow graph. All nodes that were dominated by it are also removed. This will preserve the common exit node while the back-edges of the loops that may be still present in the graph pose no problem.

An algorithmic realization can exploit a property of the modified event flow graphs that originates from the preprocessing. Our transformation ensures that the graphs contains only branching but not merging of control flow with the exception of the exit node. The only nodes left that have more than one predecessor are the ones that are the target of one or more back-edges. Therefore they have only exactly one predecessor they do not dominate:

$$\forall n, |\mathrm{pred}(n)| > 1 : |\{k \in \mathrm{pred}(n) | \neg(n \ \mathrm{dom}\ k)\}| = 1$$

As a consequence, the identified infeasible paths can be easily pruned from the modified event flow graph. For ev-

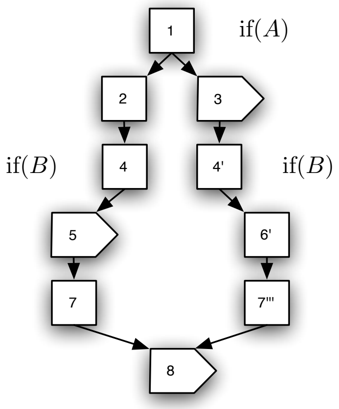

Figure 7. The pruned event flow graph

ery last node of a shortest infeasible path that was identified do the following:

- Store the set of nodes visited while traversing the graph from the start to that node. This is exactly the set of nodes that dominate the node at hand. Add the common exit node to the set.
- Remove the node and every other node that can be reached by following edges and that is not in the set defined above.
- Discard all edges that were connecting the now absent nodes.

After the post-processing, the modified event flow graph is free of any infeasible paths that were detected and can then be handled by the event dependency analysis. Following our example the result is depicted in Figure 7.

7 Example

We now provide a small and simple example to demonstrate the benefit of our idea. In the following code of an admittedly synthetic task example (Listing 2) every call of the function `send()` is meant to generate an event that triggers the next dependent task in the task graph. Exemplary execution times are annotated at the end of the code lines.

The event flow graph corresponding to the task example is depicted in Figure 8. In this case the branching conditions $A = x > 10$ and $B = x > 0$ form the implication

Table 1. Event intervals of the example

Without respecting infeasible paths:

Events	1	2	3
inI	0	6	14
startI	7	13	21
endI	2	10	16

Accounting for $A \rightarrow B$:

Events	1	2	3
inI	0	8	∞
startI	7	18	∞
endI	2	10	∞

Listing 2. Source code of the example

```
if(x > 10) {                   1
    a = x * x;                 4
    send(a);                   2
    a = x / 2;                 2
} else {
    a = -(x * x);              5
}
a = a + 1;                     1
if(x > 0) {                    1
    b = a * (a + 1);           5
} else {
    send(a);                   2
    b = a * a;                 4
}
b = b + x;                     2
send(b);                       2
clean_up();                    2
```

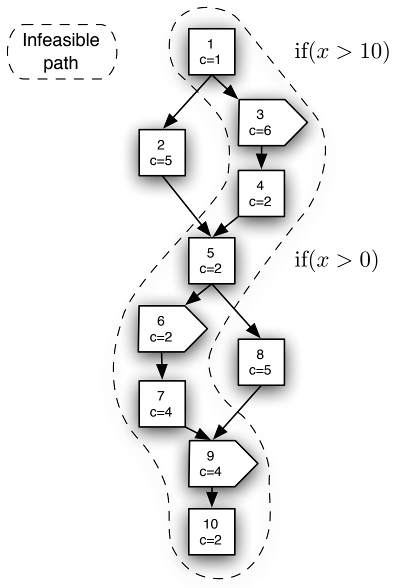

Figure 8. Event flow graph of the example

$A \rightarrow B$. There is no path between the two control block on which the the value of x is altered. This leads to an infeasible path in the control flow which is in this case $P = (1, 3, 4, 5, 6, 7, 9, 10)$. Table 1 contains the results of the relevant interval functions inI, startI and endI. As can be seen, the task generates one event less when the implication between the branching expressions is respected. Additionally, in the case of inI and startI, the density between two events has been relaxed.

8 Conclusion and future work

In this paper we have presented an approach that accounts for the pessimism which results from infeasible paths in the control flow of a task when performing an event dependency analysis. The general circumstances of infeasible paths have been defined and a method for their handling has been introduced. This method makes use of a worst-case execution time analysis tool in order to transform the flow graph of a task in a way that eliminates infeasible paths. The modified graph may then be used as the input for the event dependency analysis while reducing the pessimism that would otherwise have led to a higher event density at the output of the task. A small example was provided to show the benefit of our approach.

In future work we will attempt to extend the event dependency analysis in a way that will supersede the pre- and post-processing of the event flow graph and therefore will overcome the increase of complexity. It should be possible to use the annotated flow facts as a direct input for the analysis. A specific worst-case execution time analysis tool has to be chosen as we expect a rather tight integration of the tool and the event dependency analysis to be necessary. We will certainly have to work around specific peculiarities of any tool we may choose.

References

[1] K. Albers, F. Bodmann, and F. Slomka. Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems. In *18th Euromicro Conference on Real-Time Systems*, pages 10–106. IEEE, 2006.

[2] F. E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[3] S. K. Baruah. A general model for recurring real-time tasks. In *Real-Time Systems Symposium*, pages 114–122, 1998.

[4] R. Bodík, R. Gupta, and M. Soffa. Refining data flow information using infeasible paths. In *Software engineering-ESEC/FSE'97: 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301, page 361. Springer, September 1997.

[5] F. Bodmann, K. Albers, and F. Slomka. Analyzing the timing characteristics of task activations. In *International Symposium on Industrial Embedded Systems 2006, IES'06*, pages 1–8. IEEE, 2006.

[6] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: combining simulation with analytical estimation. *Comput. Netw.*, 41(5):641–665, 2003.

[7] 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*, 2006.

[8] 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*, pages 57–66. IEEE, 2006.

[9] R. Henia and R. Ernst. Context-aware scheduling analysis of distributed systems with tree-shaped task-dependencies. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 480–485, Washington, DC, USA, 2005. IEEE Computer Society.

[10] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1031–1052, 1997.

[11] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, 30(11):88–98, 1995.

[12] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 1(CS-TR-96-07), 1996.

[13] J. Palencia and M. González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the IEEE Real-Time Systems Symposium*, page 26. IEEE Computer Society, 1998.

[14] S. Perathoner, T. Rein, L. Thiele, K. Lampka, and J. Rox. Modeling structured event streams in system level performance analysis. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 37–46, 2010.

[15] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[16] O. Redell. Analysis of tree-shaped transactions in distributed real-time systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, pages 239–248, Washington, DC, USA, 2004. IEEE Computer Society.

[17] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models - The SymTA/S Approach.* PhD thesis, University of Braunschweig, 2005.

[18] J. Rox and R. Ernst. Construction and deconstruction of hierarchical event streams with multiple hierarchical layers. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 201–210. IEEE Computer Society, 2008.

[19] J. Rox and R. Ernst. Exploiting inter-event stream correlations between output event streams of non-preemptively scheduled tasks. In *Proc. Design, Automation and Test in Europe (DATE 2010)*, March 2010.

[20] I. Stein and F. Martin. Analysis of path exclusion at the machine code level. In *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[21] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 71–80. IEEE, 2011.

[22] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 358–363. ACM, 2006.

[23] K. Tindell. Adding time-offsets to schedulability analysis. Technical report, University of York, Computer Science Dept, YCS-94-221, 1994.

[24] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40:117–134, 1994.

[25] 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 problemoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.