Priority Assignment for Event-triggered Systems using Mathematical Programming

Martin Lukasiewycz[1], Sebastian Steinhorst[1], Samarjit Chakraborty[2]

[1] TUM CREATE, Singapore, Email: {martin.lukasiewycz, sebastian.steinhorst}@tum-create.edu.sg
[2] TU Munich, Germany, Email: samarjit@tum.de

Abstract—This paper presents a methodology based on mathematical programming for the priority assignment of processes and messages in event-triggered systems with tight end-to-end real-time deadlines. For this purpose, the problem is converted into a Quadratically Constrained Quadratic Program (QCQP) and addressed with a state-of-the-art solver. The formulation includes preemptive as well as non-preemptive schedulers and avoids cyclic dependencies that may lead to intractable real-time analysis problems. For problems with stringent real-time requirements, the proposed mathematical programming method is capable of finding a feasible solution efficiently where other approaches suffer from a poor scalability. In case there exists no feasible solution, an algorithm is presented that uses the proposed method to find a minimal reason for the infeasibility which may be used as a feedback to the designer. To give evidence of the scalability of the proposed method and in order to show the clear benefit over existing approaches, a set of synthetic test cases is evaluated. Finally, a large realistic case study is introduced and solved, showing the applicability of the proposed method in the automotive domain.

I. INTRODUCTION

Modern automobiles comprise dozens of Electronic Control Units (ECUs) and bus systems. For safety-critical functions like Anti-lock Braking System (ABS) or Electronic Stability Program (ESP) there exist hard real-time deadlines that have to be satisfied to guarantee their proper functionality. In event-triggered systems, ECUs using preemptive schedulers such as OSEK [1] or the non-preemptive Controller Area Network (CAN) bus [2] are used. In order to satisfy real-time deadlines, the schedulers use predefined priorities to determine which process or message, respectively, is scheduled in case of a contention. These priorities for all processes and messages are defined in the integration phase when the subsystems are combined to the entire system. In fact, the priorities have a very high impact on the real-time properties of functions such that they have to be chosen with deliberation. However, for realistic problems, the search space is very large and determining the priorities that satisfy all deadlines might become a very challenging task. Previous approaches are based on heuristics or do not scale well such that they are not applicable for many realistic problems. As a remedy, this paper proposes an approach using an Quadratically Constrained Quadratic Program (QCQP) such that state-of-the-art solvers can be applied efficiently.

Contributions of the paper. This paper proposes a novel method for determining priorities in event-triggered systems in order to satisfy hard real-time deadline constraints. Priority assignment is a general problem that arises in many application domains. It is a computationally hard problem and scalability is a major issue that needs to be addressed in order to make the technique relevant for many real-life problems. Our method is based on mathematical programming, relying on a graph-based approach, a translation into a QCQP, and its efficient solving. The fixed-priority scheduling problem is converted into a QCQP, taking all problem-specific constraints and the function deadlines into account. Formulations for preemptive and non-preemptive scheduling for the QCQP are proposed. This enables a modeling of OSEK operating systems and CAN buses and, thus, the application in the automotive domain becomes possible. The experimental results give evidence of the superiority of the proposed mathematical programming method in terms of scalability compared to existing approaches from literature that are based on heuristics or pruning. A realistic case study that consists of 25 ECUs and more than 300 tasks gives evidence of the applicability of the proposed method on large problems that are common in the automotive domain. Here, the proposed approach obtains a feasible priority assignment in less than a minute while previous approaches do not find any solution within 24 hours.

In case there exists no feasible priority assignment, we also propose an algorithm to determine a justification for the infeasibility. This justification is a minimal reason for the infeasibility of the problem and may be used as a feedback for the system designer or within an architectural design space exploration.

Organization of the paper. The remainder of the paper is organized as follows: Section II introduces and discusses related work. In Section III, the system model is presented. Section IV introduces the proposed methodology, including the design flow, the encoding into the QCQP, and an algorithm for the determination of a justification in case of infeasible problems. In Section V, experimental results based on synthetic test cases and a large realistic case study are given. Section VI closes the paper with concluding remarks.

II. RELATED WORK

A real-time analysis for priority-based schedulers with predefined priorities might be done efficiently using response-time analysis. A general approach for preemptive systems is presented in [3] which might be applied to ECUs with OSEK schedulers. For non-preemptive systems like the CAN bus, the method was adapted in [4] and later revised in [5].

Both approaches are applied in our paper and explained in Section III. In general, the priorities of these systems are defined in the integration phase in order to satisfy all deadlines. Other approaches like [6] that instead optimize the periods might not be applicable to control functions that are designed for specific periods only. The priorities of event-triggered systems could be optimized with population-based heuristics in order to reduce the end-to-end latencies and to satisfy the deadlines. An approach using Evolutionary Algorithms (EAs) is implemented in the SymTA/S framework and presented in [7]. However, these heuristic approaches do not perform well when the deadline constraints are hard to satisfy and, moreover, they are not able to prove the nonexistence of a feasible priority assignment. To overcome this drawback and integrate the priority assignment in an architectural decision process, an iterative pruning approach is presented in [8], [9]. In each iteration, a priority assignment is determined and analyzed such that in case of violated deadline constraints, a part of the search space is pruned. However, as presented in the experimental results, pruning infeasible solutions does not scale well and is, therefore, not applicable to large and realistic problems. An approach to tackle the priority assignment problem with an Integer Linear Program (ILP) is presented in [10] where applications cannot be defined in general but are restricted to chains of tasks and the problem has to be linearized, introducing a high number of additional constraints and variables. As a remedy, our paper proposes a novel mathematical programming method using QCQP that is able to find a feasible priority assignment efficiently, showing a very good scalability. Moreover, in case there exists no feasible priority assignment, an algorithm is proposed that can determine a minimal justification.

III. System Model

In this paper, we use a generic task graph as an application model and its mapping to an existing architecture. A task graph is defined as $G_T(T, E_T)$ where T is a set of tasks which might be processes or messages, respectively. The set of edges in the task graph E_T defines data-dependencies such that for each $(t, \tilde{t}) \in E_T$ the task $\tilde{t}$ is released (ready for execution) when the task t finished its execution. Each task in T is mapped to exactly one resource in R which might be a bus, ECU, or gateway determined by the mapping function $m : T \rightarrow R$. An example of a task graph and a mapping to an resource architecture is given in Figure 1.

To determine the worst-case response times of processes or messages, respectively, we use the recurrence equations from [3] and [5], respectively. They are of the form (1) and (2). It is assumed that the worst-case execution time e_t (equiv. to transmission time for messages) is known at design time. Moreover, each task has a period defined by h_t and a release jitter j_t.

Preemptive Scheduling. Following the approach in [3], the maximal response time r_t of a task running on a resource with a preemptive scheduler, e.g., an OSEK ECU, might be determined. Here, all possible preemptions of tasks with a higher priority defined by $hp(t)$ have to be taken into account:

$$r_t = e_t + \sum_{\tilde{t} \in hp(t)} \left\lceil \frac{r_t + j_{\tilde{t}}}{h_{\tilde{t}}} \right\rceil \cdot e_{\tilde{t}} \qquad (1)$$
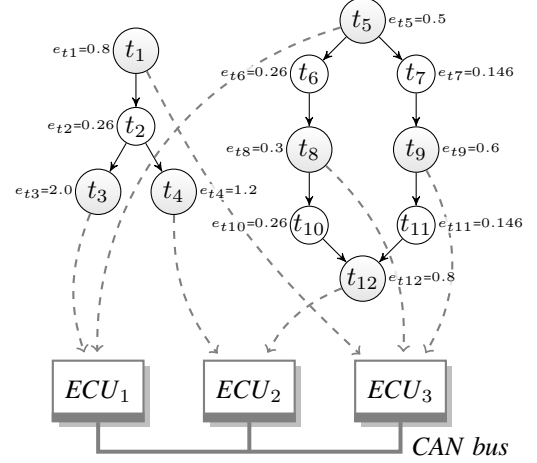

Fig. 1. Example of a task graph $G_T(T, E_T)$ consisting of two functions and a mapping to the resources R which are three ECUs and one CAN bus. Note that the tasks t_2, t_6, t_{10} are 8 byte messages and t_7, t_{11} are 2 byte messages, respectively, mapped on the CAN bus with a bandwidth of $500 kbit/s$.
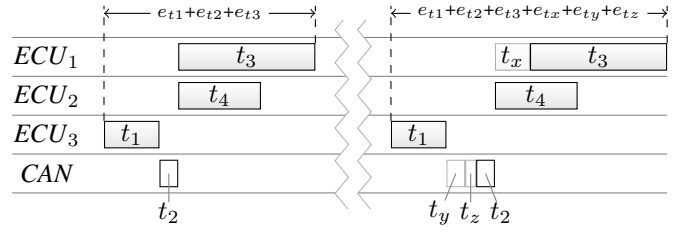

Fig. 2. Illustration of the priority-based scheduling for the first function in the example architecture in Figure 1. Here, the best-case without any preemption or suspension is illustrated as well as a case where the tasks are delayed by tasks with a higher priority, leading to a higher end-to-end delay.

The response time is defined as the sum of the execution time of the task and the execution times of all higher priority tasks. The $\left\lceil \frac{r_t + j_{\tilde{t}}}{h_{\tilde{t}}} \right\rceil$ determines how often a task $\tilde{t}$ with a higher priority might interrupt the execution of task t. The response time r_t exists on both sides of the equation, hence a fix point has to be determined. Starting with $r_t = e_t$, the recurrence equation will reach a fix point if the utilization is below or equal 1.0.

Non-Preemptive Scheduling. Corresponding to the preemptive case, a non-preemptive recurrence equation might be formulated which might be used for instance for the transmission of messages on the CAN bus. However, a higher priority task might suspend other tasks while a preemption is not possible. The recurrence equation for this case is presented in [5] and defined as follows:

$$r_t = e_t + \max_{\tilde{t} \in lp(t) \cap \{t\}} e_{\tilde{t}} + \sum_{\tilde{t} \in hp(t)} \left\lceil \frac{r_t - e_t + j_{\tilde{t}}}{h_{\tilde{t}}} \right\rceil \cdot e_{\tilde{t}} \qquad (2)$$

Here, the maximal execution time of any task with a lower priority $lp(t)$ or a previous instance of the same task has to be considered as well, since it cannot be preempted. On the other hand, the execution of the considered task cannot be preempted
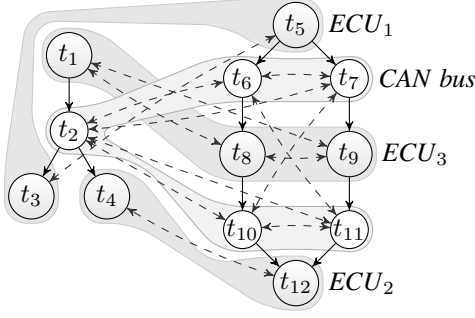
Fig. 3. Task graph $G_{TA}(T, E_{TA})$ with additional priority edges for the example architecture in Figure 1. Note that the bidirectional edges depict two independent edges between the two respective nodes. The priority edges between t_6 and t_{10} (as well as t_7 and t_{11}) might be omitted because a suspension between these messages is not possible for the given periods and deadlines.

anymore once its execution started. As a result, the amount of tasks with a higher priority that are executed before the task under consideration and might suspend this task is determined by $\left\lceil \frac{r_t - e_t + j_{\tilde{t}}}{h_{\tilde{t}}} \right\rceil$.

In a system as illustrated in Figure 1, the response times of all tasks in the task graph have to be calculated to determine whether each deadline τ_t is satisfied. Whether a deadline is violated, is determined by considering all paths to the respective task. For a part of the system from Figure 1, a schedule taking preemption and suspension into account is illustrated in Figure 2.

IV. METHODOLOGY

A. Design Flow

In order to determine the priorities of tasks, the task graph G_T is extended by additional edges E_A that define all possible priorities. One priority edge $a = (t, \tilde{t}) \in E_A$ indicates that the priority of task t is higher than the priority of task $\tilde{t}$. Therefore, these edges have to be added between all tasks that share a resource and might preempt or suspend each other. For the example in Figure 1, the resulting extended task graph $G_{TA}(T, E_{TA})$ with $E_{TA} = E_T \cup E_A$ is illustrated in Figure 3.

In an event-triggered system, a priority assignment has to be determined such that a real-time analysis guarantees that all deadlines are satisfied. In the proposed model, the priority assignment is done in a graph-based manner by reducing the set of priority edges to $E_\alpha \subseteq E_A$ such that

$$\forall (t, \tilde{t}) \in E_A : ((t, \tilde{t}) \in E_\alpha) \oplus ((\tilde{t}, t) \in E_\alpha). \quad (3)$$

Given the set of all priority edges E_A, either each priority edge or its opposite priority edge (if existent) are kept in the reduced set of priority edges E_α. Additionally, the priorities have to be transitive to enable a priority assignment using integer values. Therefore, for each three tasks that share a resource, the following requirement has to be fulfilled:

$$\forall t, \tilde{t}, \hat{t} \in T, m(t)=m(\tilde{t})=m(\hat{t}), (t, \tilde{t}), (\tilde{t}, \hat{t}) \in E_\alpha : (t, \hat{t}) \in E_\alpha \quad (4)$$

$\oplus$ is the exclusive or operator

The resulting graph $G_{T\alpha}(T, E_{T\alpha})$ with $E_{T\alpha} = E_T \cup E_\alpha$ might then be used to determine a priority assignment by considering the remaining priority edges E_α.

By adapting the graph G_{TA}, the designer might already predefine several priorities by removing edges from E_A. This might be necessary in case there exists a requirement such as a rate-monotonic scheduling. In this case, the edges $a = (t, \tilde{t}) \in E_A$ with $h_t > h_{\tilde{t}}$ have to removed from E_A. Correspondingly, a requirement for deadline-monotonic scheduling might be formulated. These additional requirements help to reduce the search space significantly, complying with the state-of-the-art design in many domains. In the experimental results, for instance, rate-monotonic scheduling is assumed if the periods of tasks are not equal. Note that further domain knowledge might be incorporated such as illustrated in Figure 3. Here, there exist no priority edges between t_6 and t_{10} (as well as t_7 and t_{11}). This is done to prohibit cycles and because a suspension between these messages is not possible for the given periods and deadlines.

B. Encoding

In the following, the problem of priority assignment is defined as Quadratically Constrained Quadratic Program (QCQP). In contrast to ILPs, a QCQP might also contain terms that are quadratic, considering products of two variables. This is required here in the Constraints (11) and (13) that model the recurrence Equations (1) and (2). In case the QCQP is convex (as it is the case in the provided formulation), modern solvers [11] that transform the problem into a Second-Order Cone Program (SOCP) [12] can determine a feasible priority assignment efficiently using an interior point method.

Priority Assignment. In order to determine the desired set of active priority edges E_α from the set E_A, a binary variable $\mathbf{a} \in \{0, 1\}$ is introduced for each edge $a \in E_A$. This binary variable determines whether the edge is in E_α (1) or not (0). The requirements on the set E_α as stated in (3) and (4) are summarized in the following constraints:

$\forall a = (t, \tilde{t}), \tilde{a} = (\tilde{t}, t) \in E_A :$

$$\mathbf{a} + \tilde{\mathbf{a}} = 1 \quad (5)$$

$\forall a = (t, \tilde{t}) \in E_T, \neg \exists \tilde{a} = (\tilde{t}, t) \in E_A :$

$$\mathbf{a} = 1 \quad (6)$$

$\forall a = (t, \tilde{t}), \tilde{a} = (\tilde{t}, \hat{t}), \hat{a} = (\hat{t}, t) \in E_A :$

$$\mathbf{a} + \tilde{\mathbf{a}} + \hat{\mathbf{a}} \leq 2 \quad (7)$$

Constraint (5) and (6), respectively, fulfill the requirement in (3). The first constraint considers the case that there exist priority edges in both directions between two tasks, the latter constraint considers the case that there exists only a single priority edge in one direction between two tasks. Constraint (7) fulfills (4) by excluding all combinations of priority assignments that violate the transitive requirement, i.e., in case a cycle of priority edges exists between three tasks. These constraints ensure that cycles in the resulting task graph $G_{T\alpha}$ on tasks that are mapped to the same resource do not exist. An example for such as a cycle is given as π_1 in Figure 4.

Global Cycles. Cycles across different functions as illustrated as π_2 in Figure 4 and discussed in [13] might become a burden

$\pi_2 = (t_1, t_2, t_3, t_5, t_6, t_8)$
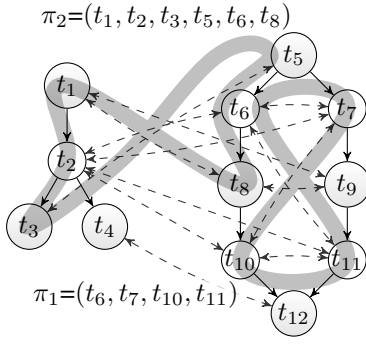
$\pi_1 = (t_6, t_7, t_{10}, t_{11})$

Fig. 4. Illustration of possible cycles in the $G_{T\alpha}$ that results from G_{TA}. The cycle π_1 is affecting tasks that are mapped to a single resource, the cycle π_2 is distributed over two functions.

when determining the real-time properties of systems. Therefore, it is useful to exclude these cycles from the search space by introducing additional variables and constraints. Although it is not necessary to exclude these cycles, it minimizes the complexity of the system and helps in reducing the search space. It becomes necessary to add an additional variable $\mathbf{c_t} \in \mathbb{R}$ for each $t \in T$ that is used as counter. For each used edge (either in E_T or E_α), the source task has a lower counter value than the destination task, leading to an exclusion of all cycles. For both types of edges, the constraints are defined as follows:

$\forall (t, \tilde{t}) \in E_T :$
$$\mathbf{c_t} < \mathbf{c_{\tilde{t}}} \tag{8}$$

$\forall a = (t, \tilde{t}) \in E_A :$
$$\mathbf{c_t} < \mathbf{c_{\tilde{t}}} + |T| - |T| \cdot \mathbf{a} \tag{9}$$

Constraint (8) ensures that the counter values fulfill the requirement for all data dependencies. Constraint (9) states that for each priority edge the counter values have to be incremented only if the edge is active, i.e., in E_α.

Additionally, it might be necessary to assign implications on priority edges. For instance, if messages are routed over different components which results in multiple tasks in T. Then, the messages should not change the their priorities for different components as expressed by the following constraint:

$\forall a = (t, t'), \tilde{a} = (\tilde{t}, \tilde{t}') \in E_T, t \stackrel{prio}{=} \tilde{t}, t' \stackrel{prio}{=} \tilde{t}' :$
$$\mathbf{a} = \mathbf{\tilde{a}} \tag{10}$$

Preemptive Scheduling. Given the priority edges and variables, respectively, it is possible to determine the response times of tasks. For each task t, the response time is defined as the variable $\mathbf{r_t} \in \mathbb{R}$. Additionally, it is necessary to introduce the release jitter $\mathbf{j_t} \in \mathbb{R}$ for each task. To encode how often a task with a higher priority might preempt the execution, the variable $\mathbf{i_a} \in \mathbb{N}$ is introduced for each priority edge. The response time for a preemptive resource is determined by the following constraints:

$\forall t \in T :$
$$\mathbf{r_t} = e_t + \sum_{a = (\tilde{t}, t) \in E_A} e_{\tilde{t}} \cdot \mathbf{a} \cdot \mathbf{i_a} \tag{11}$$

$\forall t \in T, a = (\tilde{t}, t) \in E_A :$
$$\mathbf{i_a} \geq \frac{1}{h_{\tilde{t}}} \cdot \mathbf{r_t} + \frac{1}{h_{\tilde{t}}} \cdot \mathbf{j_{\tilde{t}}} \tag{12}$$

Constraint (11) corresponds to Equation (1) where the variable $\mathbf{a}$ defines whether some other task has a higher priority and $\mathbf{i_a}$ corresponds to the amount of preemptions that are possible by another task. The value of $\mathbf{i_a}$ is determined by Constraint (12). Note that although the variable for the number of preemption is open to the top and the response time might be over-approximated using these constraints, the values are never higher than necessary to fulfill the deadline constraints. **Non-Preemptive Scheduling.** Corresponding to the preemptive schedule, the constraints for the non-preemptive scheduling are defined in the following. The formulation requires the additional variable $\mathbf{b_t} \in \mathbb{R}$ that encodes the maximal delay by a lower priority task or a previous instance of the task itself. The constraints are formulated as follows:

$\forall t \in T :$
$$\mathbf{r_t} = e_t + \mathbf{b_t} + \sum_{a = (\tilde{t}, t) \in E_A} e_{\tilde{t}} \cdot \mathbf{a} \cdot \mathbf{i_a} \tag{13}$$

$$\mathbf{b_t} \geq e_t \tag{14}$$

$\forall t \in T, a = (t, \tilde{t}) \in E_A :$
$$\mathbf{b_t} \geq \mathbf{a} \cdot e_{\tilde{t}} \tag{15}$$

$\forall t \in T, a = (\tilde{t}, t) \in E_A :$
$$\mathbf{i_a} \geq \frac{1}{h_{\tilde{t}}} \cdot \mathbf{r_t} + \frac{1}{h_{\tilde{t}}} \cdot \mathbf{j_{\tilde{t}}} - \frac{e_t}{h_{\tilde{t}}} \tag{16}$$

Constraint (13) corresponds to Equation (2). Constraints (14) and (15), respectively, determine the maximal delay due to a previous instance of the task itself or a task with lower priority. Constraint (16) determines how often a higher priority task might suspend the considered task.

Deadlines. In order to determine whether each deadline is satisfied, it necessary to calculate the delay and release jitter for each task along each path in the task graph. For each task t, the delay is defined as variable $\mathbf{d_t} \in \mathbb{R}$ and the release jitter is defined as $\mathbf{j_t} \in \mathbb{R}$. The constraints to determine the values of these variables and to determine whether the deadlines are fulfilled are as follows:

$\forall t \in T, \neg \exists (\tilde{t}, t) \in E_T :$
$$\mathbf{d_t} = \mathbf{r_t} \tag{17}$$
$$\mathbf{j_t} = 0 \tag{18}$$

$\forall t \in T, (\tilde{t}, t) \in E_T :$
$$\mathbf{d_t} \geq \mathbf{r_t} + \mathbf{d_{\tilde{t}}} \tag{19}$$
$$\mathbf{j_t} \geq \mathbf{r_{\tilde{t}}} - e_{\tilde{t}} + \mathbf{j_{\tilde{t}}} \tag{20}$$

$\forall t \in T :$
$$\mathbf{d_t} \leq \tau_t \tag{21}$$

For each task that has no predecessor, the delay and jitter are defined by the Constraints (17) and (18), respectively. Here, the initial release jitter is assumed to be 0. In case a task has predecessors, the delay and jitter values are determined by the Constraints (19) and (20), respectively. Here, the delay is determined as the sum of the maximum of the delays of the preceding tasks and the response time of the current task. Correspondingly, the jitter is the maximum of the preceding task jitter values and the current jitter which is the difference between the worst-case response time and the execution time. Finally, Constraint (21) ensures that all deadlines are satisfied by constraining the delays.
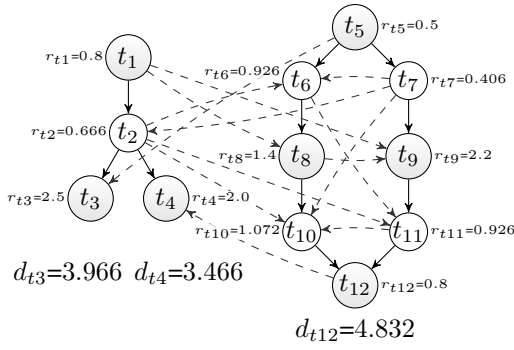
Fig. 5. Task graph $G_{T\alpha}$ that corresponds to the feasible priority assignment for the system in Figure 1 and Figure 3.
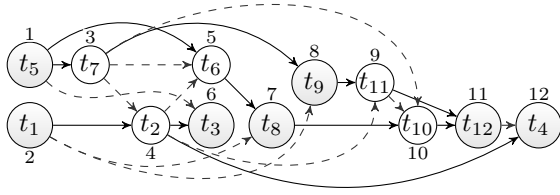

Fig. 6. Topologically ordered graph with annotated priorities for $G_{T\alpha}$ in Figure 1.

For the example in Figure 1, the QCQP determines the solution that corresponds to the task graph $G_{T\alpha}$ in Figure 5. Note that already for this small example, there exist 248 different priority assignments while only two of them satisfy the deadlines. A global priority assignment is obtained by ordering the graph $G_{T\alpha}$ topologically as illustrated in Figure 6.

C. Justification

In case there exists no feasible priority assignment that satisfies the deadlines, the QCQP will terminate without a feasible result. In this case, it is desirable to analyze the reason for the infeasibility of the problem. This is also known as the determination of the *justification* and used in mathematical programming known also as Irreducible Infeasible Subsets (IIS) [14] or Minimally Unsatisfiable Subformula (MUS) [15], respectively. Here, a justification equals a subgraph of G_{TA} that is still unsatisfiable and becomes satisfiable when a single task is removed from this subgraph. Removing a task equals the setting of its execution time to 0. In the following, a problem-specific approach is presented.

In case a task graph G_{TA} is infeasible, the justification is determined by Algorithm 1.

The algorithm iteratively determines the tasks that can be deleted (denoted as $\tilde{T}$) from the task graph G_{TA} while keeping it infeasible. The algorithm starts with an empty set $\tilde{T}$ (line 1) and iterates over all tasks in the graph (line 2). For each task, the execution is set to 0 while the original value is saved in the temporary variable x (line 3-4). Setting the execution time to 0 equals the deletion of the corresponding task while all the deadlines are still considered. In case the task graph G_{TA} becomes feasible after the deletion of task t, the execution time is reset to its original value (line 5-6). If the task graph remains infeasible, the task t is not part of the justification

Algorithm 1 Algorithm for the determination of a justification for an infeasible task graph G_{TA}.

1: $\tilde{T} = \{\}$
2: **for** $t \in T$ **do**
3:     $x = e_t$
4:     $e_t = 0$
5:     **if** G_{TA} is feasible **then**
6:         $e_t = x$
7:     **else**
8:         $\tilde{T} = \tilde{T} \cup \{t\}$
9:     **end if**
10: **end for**
11: **return** $G_{TA}(T\backslash\tilde{T}, \{(t,\tilde{t})|(t,\tilde{t}) \in E_{TA} \wedge t,\tilde{t} \notin \tilde{T}\})$
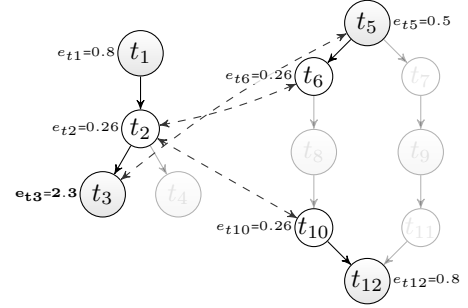

Fig. 7. Determined justification in case the execution time of task t_3 is increased to 2.3 and the problem becomes infeasible.

and is added to the set of deleted tasks (line 8). Finally, the justification is determined by removing the tasks $\tilde{T}$ from the task graph (line 11).

An example for the determination of a justification for the system in Figure 1 is given in the following. If the execution time of task t_3 is increased to 2.3, the problem becomes infeasible, i.e., there exists no feasible priority assignment such that all deadlines are satisfied. The justification determined by the presented algorithm is given in Figure 7 which shows that only 7 tasks have to considered to understand the reason for infeasibility.

V. EXPERIMENTAL RESULTS

In the following, experimental results are presented that give evidence of the applicability of the proposed approach. For this purpose, a set of synthetic test cases is evaluated and a large realistic case study from the automotive domain is introduced. All experiments were carried out on an Intel Xeon QuadCore CPU with 3.20 GHz and 12 GB RAM using the GUROBI QCQP solver [11].

Scalability. In order to show the scalability of the proposed approach and to compare it with existing approaches, a set of 100 synthetic test cases is used. The test cases where randomly generated and have various complexities. The smallest test case consists of 2 ECUs, 1 CAN bus, and 2 functions with 11 tasks. The largest test case consists of 23 ECUs, 4 CAN buses connected via a central gateway, and 25 functions resulting in 294 tasks. For all test cases, tight deadlines exist that make only a small fraction of the entire search space feasible.

The proposed approach is compared to the *pruning* approach from [9] and to an Evolutionary Algorithm (EA) method
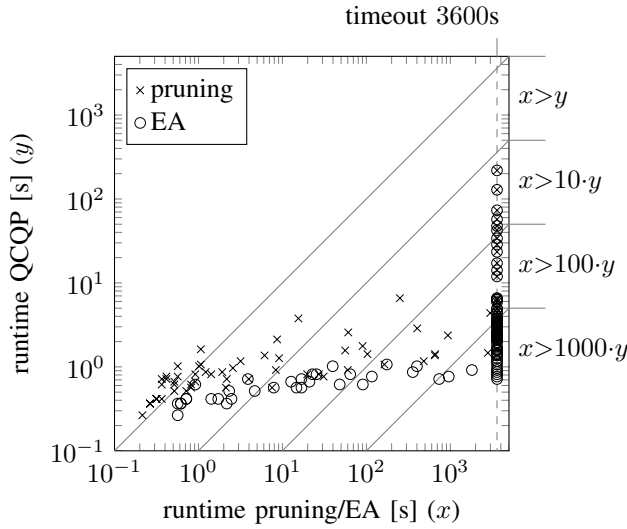
Fig. 8. Comparison of the proposed method (QCQP) with existing approaches on a set of 100 test cases. Note that the pruning approach failed to deliver results for 42 test cases and the EA for 67 test cases due to a predefined timeout of 3600 seconds.

cluster (CAN bus)	ECUs	tasks	functions / periods
body	6	71	40,40,50,80,80,80
driver assistance systems	3	74	5,10,20,40,80,80
chassis	4	97	10,20,40,40,50,50,100
infotainment	4	38	5,80,100
powertrain	8	52	5,10,40,50,50,80
system	25	332	

TABLE I
CLUSTERS (BUSES), ECUS, NUMBER OF TASKS, AND PERIODS OF THE
FUNCTIONS OF THE CASE STUDY.

seconds.

VI. CONCLUDING REMARKS

This paper presented an exact approach for the determination of priorities in event-triggered systems such as automotive systems based on OSEK schedulers and CAN buses. For this purpose, the problem of priority assignment is converted into a QCQP, using a state-of-the-art solver. This proposed approach was extended to determine one minimal justification in case the problem does not permit a feasible solution. This gives the designer a feedback regarding the reason of the infeasibility. The experimental results show the superiority of the proposed method compared to existing approaches.

In future work, the approach will be extended by an additional determination and consideration of offsets. Moreover, the justification determination will be improved regarding the runtime and consequently integrated into an architectural design space exploration.

corresponding to [7] that repairs the solutions in order to avoid cycles. The runtimes with all approaches are illustrated in Figure 8. The proposed approach is denoted as QCQP and is superior to the other approaches. In particular, for the big test cases, the QCQP approach is multiple orders of magnitude faster than the other approaches. It is capable of finding a feasible solution for all 100 test cases within 3600 seconds while the pruning approach failed to deliver feasible solutions for 42 test cases and the EA failed in 67 cases.

Case Study. In order to give evidence of the applicability of the proposed methodology on realistic problems, a case study is introduced. The case study consists of five CAN buses that define clusters and a central gateway that interconnects these clusters. Note that the functions in each cluster are not solely mapped to ECUs of this cluster such that it is not possible to determine the priorities for each cluster separately. The details of the case study are summarized in Table I that also shows the periods of the functions in each cluster. Overall, the case study consists of 25 ECUs and 27 functions that result in 332 tasks with 2583 priority edges in the set E_A (several edges might be removed due to the rate-monotonic requirement).

With the proposed methodology, it is possible to determine a feasible priority assignment within 44.3 seconds while other approaches (pruning and EA) do not find a feasible solution within 24 hours. This again shows the superiority of the proposed approach and its ability to cope with realistic problems.

In order to give evidence of the capabilities of the justification determination, the execution times of half of the tasks are increased by 0.1ms to make the case study infeasible. The determination of the justification requires 514 seconds and results in 74 tasks from 24 functions and all five clusters. In case the execution times of two third of the tasks are increased by 0.1ms or 0.2ms, respectively, the justification becomes significantly smaller. It results in 7 tasks from one cluster and one single function and is determined within 354

REFERENCES

[1] OSEK, "OSEK VDX Portal," http://www.osek-vdx.org/.
[2] CAN, "Controller Area Network," http://www.can.bosch.com/.
[3] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[4] K. Tindell, A. Burns, and A. Wellings, "Calculating Controller Area Network (CAN) Message Response Times," *Control Engineering Practice*, vol. 3, pp. 1163–1169, 1995.
[5] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.
[6] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period Optimization for Hard Real-time Distributed Automotive Systems," in *Proc. of DAC*, 2007, pp. 278–283.
[7] A. Hamann, M. Jersak, K. Richter, and R. Ernst, "Design Space Exploration and System Optimization with SymTA/S–Symbolic Timing Analysis for Systems," in *Proc. of RTSS*, 2004, pp. 469–478.
[8] F. Reimann, M. Glaß, C. Haubelt, M. Eberl, and J. Teich, "Improving Platform-based System Synthesis by Satisfiability Modulo Theories Solving," in *Proc. of CODES+ISSS*, 2010, pp. 135–144.
[9] F. Reimann, M. Lukasiewycz, M. Glass, C. Haubelt, and J. Teich, "Symbolic System Synthesis in the Presence of Stringent Real-time Constraints," in *Proc. of DAC*, 2011, pp. 393–398.
[10] B. Lisper and P. Mellgren, "Response-time Calculation and Priority Assignment with Integer Programming Methods," in *Proc. of ECRTS*, 2001, pp. 13–16.
[11] Gurobi Optimizer, "Gurobi 5.0," http://www.gurobi.com/.
[12] H. Mittelmann, "An Independent Benchmarking of SDP and SOCP Solvers," *Mathematical Programming*, vol. 95, no. 2, pp. 407–430, 2003.
[13] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi, "Cyclic Dependencies in Modular Performance Analysis," in *Proc. of EMSOFT*, 2008, pp. 179–188.
[14] O. Guieu and J. W. Chinneck, "Analyzing Infeasible Mixed-Integer and Integer Linear Programs," *INFORMS Journal on Computing*, vol. 11, pp. 63–77, 1999.
[15] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. Markov, "Amuse: a minimally-unsatisfiable subformula extractor," in *Proc. of DAC*, 2004, pp. 518–523.