ABSTRACT

Due to the ever increasing system complexity, deciding whether a given platform is sufficient to implement a set of applications under given constraints becomes a serious bottleneck in platform-based design. As a remedy, the work at hand proposes a novel automatic platform-based system synthesis procedure, inspired by techniques developed in the context of automatic system verification known as Satisfiability Modulo Theories. It tightly couples the computation of a feasible allocation and binding with nonfunctional constraint checking where, in contrast to existing approaches, not only linear constraints but even nonlinear constraints are supported. This allows to efficiently prove whether there exists a feasible implementation of a set of applications on the given platform with respect to both, functional and nonfunctional constraints. Moreover, an approach for early learning based on feasibility checking of partial implementations is proposed that can significantly improve the synthesis runtime, especially in case the selected platform imposes stringent constraints on the implementation. The effectiveness of this approach is shown for an automotive ECU network design that requires Modular Performance Analysis to ensure non-functional nonlinear timing constraints.

Categories and Subject Descriptors
C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms
Design, Algorithms

1. INTRODUCTION

At the Electronic System Level (ESL), the task of synthesis consists of (1) allocating processors, hardware accelerators, network controllers, memories, etc., (2) binding tasks onto computational resources as well as routing control and data dependencies among tasks on the communication architecture, and (3) scheduling computation and communication. Due to the ever increasing system complexity (including both, application and architecture complexity) and stringent nonfunctional constraints, e.g., timing constraints, power and volume consumption constraints, system synthesis is still a challenging task. In the past, there has been a trend towards platform-based design [12] which reduces architectural complexity by defining an architectural template also called a platform. This platform consists of a set of resources for computation and communication significantly reducing the size of the design space. Using platform-based design, system synthesis can be formally defined as a selection and assignment problem, i.e., allocation and binding. The corresponding combinatorial multi-objective optimization problem is known as design space exploration. Many system level design methodologies have been developed upon this idea in the past decade [1, 5, 13, 19, 21]. Most of these methodologies are based on a common scheme [8] where design space exploration is a twofold task: (1) searching the design space and (2) successively evaluating objectives and checking nonfunctional constraints.

As application complexity grows, constraints imposed on the implementation become more and more stringent. In particular, deciding if a set of applications can be implemented on a given platform is a hard and currently often even unsolvable task, especially in the presence of nonfunctional nonlinear constraints. Several approaches integrating linear constraint checking into allocation and binding computation have been proposed in the past. Unfortunately, many constraints in embedded system design are nonlinear and relaxations often lead to false analysis results. Thus, existing system synthesis approaches struggle with the problem of producing many infeasible implementations. As a consequence, identifying infeasible regions of the design space as early as possible would directly contribute to the overall performance of a system synthesis approach.

In this paper, we propose a novel automatic platform-based system synthesis procedure which is able to identify infeasible regions of the design space which are due to nonlinear constraints. For the first time, this allows to efficiently prove whether there exists a feasible implementation of a set of applications on the given platform with respect to both, linear and nonlinear constraints. For this purpose, we adopt methods developed in the context of formal system verification. These methods are known as SMT solvers (Satisfiability Modulo Theories) and have been successfully applied to the verification problem of hardware components above the RTL (Register Transfer Level) and even formal software verification. Hence, we call our proposed method SMT-based System Synthesis (S3). Moreover, an approach for early learning based on feasibility checking of partial implementations with respect to nonfunctional constraints during system synthesis is proposed. In particular, the im-
plementing as an indirect SMT approach allows to seamless apply any external analysis tool, e.g., for performance, real-time, or power analysis for the check of nonfunctional constraints. As 3S is, thus, able to identify infeasible regions earlier than previously developed approaches, an acceleration of the system synthesis could be seen. Especially in the context of design space exploration, 3S significantly shortens the exploration time.

The remainder of the paper is organized as follows: Related work is discussed in Section 2. In Section 3 a formal problem definition is provided and motivating examples are given. The proposed SMT-based System Synthesis (3S) is introduced and comprehensively discussed in Section 4, before experimental results are presented in Section 5. Conclusions are drawn in Section 6.

2. RELATED WORK

In the past decade, there has been a trend towards platform-based design, cf. [12]. It significantly reduces architectural complexity by defining a so called platform. Today nearly all system level synthesis tools are based on this principle, cf. [1, 5, 15, 16, 17, 21]. Platform-based design allows to separate the system synthesis task into a combinatorial decision problem, i.e., computing allocation, binding, and scheduling, and performance evaluation including feasibility checking. In order to handle huge design spaces, approaches that use symbolic representations of the design space have been shown to be very successful in solving the combinatorial decision problem. Hence, in the following, the discussion of related work is limited to such approaches.

Early work in the area of hardware/software partitioning is based on solving integer linear programming as, e.g., presented by Niemann and Marwedel in [17]. In [3], Blickle et al. have shown that the Boolean satisfiability problem could be reduced in polynomial time to the problem of computing feasible allocations and bindings in platform-based system synthesis approaches, i.e., computing a feasible allocation and binding is an NP-complete problem. Haubelt et al. used the reverse result to define a SAT-based system synthesis approach [10]. An analogous approach based on binary decision diagrams is presented by Neema in [16]. As the required amount of memory for storing the binary decision diagrams may grow exponentially in the number of Boolean variables, the latter approach could only be applied to small systems. Lukasiewycz et al. reported a first approach to integrate nonfunctional constraint checking in the SAT-based allocation and binding computation in [15]. However, this approach is restricted to linear constraints as they are directly represented in the Boolean formula. Furthermore, it uses a binary encoding of integer variables which leads to an exponential grow for the required number of variables. The proposed approach in the work at hand extends all the above mentioned approaches by integrating background theories into the SAT-based allocation and binding computation. That way it is possible to check nonfunctional nonlinear constraints during allocation and binding computation.

SAT-solvers have been successfully applied to the task of system verification in the past, cf. [20]. Thereby, the success is at least partially build on the improvements in SAT-solver technology in the 1990s. In order to improve applicability from SAT-based hardware verification approaches from logic level to higher levels and even software and system verifications SMT-solvers have been developed. In hardware verification the goal is to preserve information of high-level data types or arithmetic operations without the need to use equivalent, but bigger bit-level representatives. In software and system verification, abstraction is the key motivation, in order to handle device drivers or huge memories. For all of these purposes, SMT solvers integrate appropriate background theories. Examples of background theories are ”Equality and Uninterpreted Functions”, ”Linear Arithmetic for Integers”, and ”Bitvectors”.

SMT-solvers in general can be divided in two classes, direct and indirect SMT-solvers. Direct SMT-solvers translate the constraints given in a background theory into Boolean formulas. Afterwards a standard SAT-solver is used to find a satisfying variable assignment. A drawback of direct SMT-solvers results from the extreme variable blowup. In contrast, indirect SMT-Solvers rely on available background theorem provers. Examples of state-of-the-art SMT-solvers are Barcelogic [18], CVC4Lite/CVC3 [2], UCLID [14], and Yices [6].

In summary, to the best of authors’ knowledge, no seamless integration of nonfunctional constraint checking with SAT-based allocation and binding computation in platform-based design has been reported before. Furthermore, the authors are not aware of any indirect SMT-solver implementation integrating Modular Performance Analysis as background theory as will be presented in the Sec. 5.

3. PROBLEM STATEMENT AND MOTIVATING EXAMPLES

In Platform-Based Design (PBD), a set of applications is mapped onto a given platform. The resulting implementation has to satisfy a set of functional and nonfunctional constraints. To formalize PBD, a mathematical notation is introduced first.

Given a set of applications represented as an application graph \( g_A = (P, D) \) and a platform represented as platform graph \( g_P = (R, C) \). Each vertex \( p \in P \) models a process and edges \( d \in D \subseteq P \times P \) represent control or data dependencies. Vertices \( r \in R \) model platform resources, e.g., processors, memories, buses, etc., and edges \( c \in C \subseteq R \times R \) model connections between resources. In order to model the ability of intra-resource communication, self-loops can be defined, i.e., \( (r, r), (r, r) \in C \). Beside the application graph \( g_A \) and the platform graph \( g_P \), a set of mapping constraints \( M \subseteq P \times R \) is given. Each mapping constraint \( m = (p, r) \in M \) indicates a possible mapping of a process \( p \) onto a resource \( r \).

A subtask of system synthesis is to compute an allocation \( A \subseteq R \), a binding \( B \subseteq M \), and a schedule \( T \). An allocation \( A \) is a subset of platform resources used in the implementation. A binding \( B \) is a subset of mapping constraints assigning processes to resources. The schedule can be static (predefined start times for the processes) or dynamic (assigning priorities, deadlines, etc., processes).

An implementation \( x = (A, B, T) \) in \( X \) is one possible solution in the search space \( X = 2^R \times 2^M \times 2^T \). Any feasible implementation has to obey functional \( C_F \) and nonfunctional constraints \( C_N \). Thus, PBD can be formally defined as:

**Problem.** Find an implementation \( x \in X_f \subseteq X \) with \( X_f = X_F \cap X_N \) being the set of feasible implementations as an intersection of the implementations \( X_F \) that fulfill the given functional constraints \( C_F \) and \( X_N \) that fulfill the given nonfunctional constraints \( C_N \).

Functional constraints typically given in PBD impose that

1. each resource used for binding a process is allocated, i.e., \( \forall p, r \in B : r \in A \),

2. each process is bound exactly once, i.e., \( \forall p \in P : |\{m \in B \mid m = (p, r)\}| = 1 \), and

3. each control or data dependency can be routed on the allocated architecture, i.e., \( \forall (p_i, p_j) \in D \cup B : (p_i, r_i), (p_j, r_j) \in B : (r_i, r_j) \in C \).

Any pair of allocation and binding fulfilling these constraints are called feasible with respect to the functional constraints.
Note that each implementation \(x \in X_F\) and, hence, \(x \in X_f\) is based on such a feasible allocation and binding. Moreover, often computing a schedule \(T\) can be expressed as a combinatorial decision problem. As this, however, is highly problem-dependent, we will limit the following presentation to the computation of allocations and bindings.

The task of computing a feasible allocation and binding with respect to functional constraints can be formulated as a Boolean satisfiability problem (SAT). For this purpose, a characteristic function \(\Psi_F : 2^\Theta \rightarrow \{0, 1\}\) of \(X_F\) is defined that encodes all implementations that are feasible with respect to functional constraints with \(\Theta\) being the set of all decision variables, i.e.,

\[
\Theta = \Theta_R \cup \Theta_C \cup \Theta_M, \quad \text{with}
\]

\[
\Theta_R = \{ \rho_i | \forall r \in R : \rho_i \in \{0, 1\} \},
\]

\[
\Theta_C = \{ \gamma_{ij} | \forall (r_i, r_j) \in C : \gamma_{ij} \in \{0, 1\} \},
\]

\[
\Theta_M = \{ \mu_{ij} | \forall (p_i, r_j) \in M : \mu_{ij} \in \{0, 1\} \}.
\]

Here, the activation of a resource \(r_i \in R\) is encoded by variable assignment \(\rho_i := 1\), the availability of a connection \((r_i, r_j) \in C\) between resources in an allocation is encoded by \(\gamma_{ij} := 1\), and the selection of a mapping possibility \(m = (p_i, r_j) \in M\) is encoded by the variable assignment \(\mu_{ij} := 1\). Moreover, the relation \(\sigma : 2^X \rightarrow \{1, 0\}^{[\Theta]}\) is given that returns a unique encoding of each implementation \(x \in X\) using the specified variables in \(\Theta\). Now, allocation and binding can be expressed by the encoding of resource activation and mapping selection:

\[
A = \{ r_i \in R | \rho_i = 1 \} \quad (1)
\]

\[
B = \{ (p_i, r_j) \in M | \mu_{ij} = 1 \} \quad (2)
\]

By using the above encodings, the characteristic function \(\Psi_F\) looks as follows:

\[
\Psi_F = \forall (p_i, r_j) \in M : \mu_{ij} \rightarrow p_j \quad (3a)
\]

\[
\land \forall (r_i, r_j) \in C : \rho_i \land \rho_j \rightarrow \gamma_{ij} \quad (3b)
\]

\[
\land \forall p_i \in P : \bigvee_j \mu_{ij} \land \bigwedge_{j \neq k} \mu_{ij} \land \mu_{ik} \quad (3c)
\]

\[
\land \forall d = (p_i, p_j) \in D : \mu_{ik} \land \mu_{jl} \rightarrow \gamma_{kl} \quad (3d)
\]

Equation (3a) implies\(^1\) that each selected mapping possibility ends on an allocated resource. Using logical equality\(^2\), Eq. (3b) ensures that an existing connection between two resources is allocated if and only if the two incident resources are allocated. The requirement that each process is bound exactly once is given in Eq. (3c) by requiring at least one mapping possibility to be selected and forbidding the selection of any pair of mapping possibility for a process. Finally, Eq. (3d) imposes that communicating processes are bound to adjacent resources in the allocation. Any consistent variable assignment such that the characteristic function \(\Psi_F\) evaluates to 1 represents a feasible allocation and binding with respect to functional constraints.

As will be discussed in detail in the next section, existing approaches are restricted to the characteristic function \(\Psi_F\). However, to solve the system synthesis problem correctly, functional as well as nonfunctional constraints need to be respected. Thus, the problem targeted in the work at hand can be refined as follows:

**Problem.** Find a variable assignment \(\vartheta\) such that \(\Psi_F = 1\) with \(\Psi_F = \Psi_F \land \Psi_N\) being the characteristic function of the set of feasible implementations encoded as the conjunction of the characteristic functions \(\Psi_F\) and \(\Psi_N\).

\[^1\]a \rightarrow b \iff \neg b \lor a
\[^2\]a \leftrightarrow b \iff (a \rightarrow b) \land (b \rightarrow a)

---

Figure 1: Application consisting of five processes and a platform containing two processors connected by a bus.

\(\Psi_F\) encodes all implementations that fulfill the given functional constraints \(C_F\) while \(\Psi_N\) encodes all implementations that fulfill the given nonfunctional constraints \(C_N\).

In this work, an approach based on indirect Satisfiability Modulo Theories (SMT) solving is presented that efficiently learns the characteristic function \(\Psi_F\), based on the subsequent construction of \(\Psi_N\). In particular, we present:

- An early learning approach based on feasibility checking of partial implementations that allows to speed up the search for feasible implementations by pruning whole sets of implementations that are infeasible due to nonfunctional nonlinear constraints from the search space.

- For the first time, the ability to efficiently prove whether there exists a feasible implementation of a set of applications on a given platform by incorporating symbolic techniques instead of using costly exhaustive search approaches.

- The seamless integration of external analysis approaches, e.g., for performance, real-time, or power analysis for checking nonfunctional is shown. Thus, existing system synthesis approaches that incorporate several external analysis tools may also benefit from the proposed formal technique.

Throughout the paper, the following running example shall serve as a motivation and an illustration of the proposed technique:

**Example** An application consisting of five chained processes \(p_1, \ldots, p_5\) is given, cf. Figure 1. The available platform consists of two processors \((r_1, r_2)\) connected by a bus \((r_3)\). Both processors \(r_1\) and \(r_2\) support intra-processor communication. Processes \(p_1, p_3,\) and \(p_5\) are computational processes and can be implemented either on \(r_1\) or \(r_2\). Processes \(p_2\) and \(p_4\) are communication processes and can be implemented on any resource \(r_1, \ldots, r_3\). A simplified Boolean formula representing all implied inclusions that are feasible with respect to functional constraints is given by the characteristic function \(\Psi_F:\)

\[
\Psi_F = (\mu_{11} \lor \mu_{12}) \land (\mu_{31} \lor \mu_{32}) \land (\mu_{51} \lor \mu_{52}) \land (\mu_{11} \lor \mu_{21} \lor \mu_{12} \lor \mu_{22} \lor \mu_{13} \lor \mu_{23})
\]

\[
\land (\mu_{31} \lor \mu_{32} \lor \mu_{33} \lor \mu_{34} \lor \mu_{35} \lor \mu_{36} \lor \mu_{37})
\]

where \(\oplus\) denotes the exclusive or operator.

A standard SAT-solver can be used to find a satisfying variable assignment to \(\Psi_F\), i.e., \(\exists \vartheta : \Psi_F\). The result is a feasible allocation \(A\) and binding \(B\) as given in Eq. (1) and (2), if such a solution exists.

\(^3\)The allocation of resources is implied by the binding, i.e., whenever a process is bound on a resource, its activation variable is set to 1.
to a nonfunctional constraint. As the characteristic function \( \Psi_f \) only represents functional constraints, the SAT-solver will assume the implementation to be feasible. To overcome this problem, linear constraints can be encoded by Boolean formulas and conjunctively connected to \( \Psi_f \). Unfortunately, this approach does not succeed in the presence of nonlinear constraints as a relaxation of nonlinear constraints to linear constraints and successively encoding may lead to false results.

As a remedy, an approach is proposed where a state-of-the-art SAT-solver is hybridized with a set of external evaluator functions \( G \) that are responsible to analyze an implementation and determine whether nonfunctional nonlinear constraints are met. This can be interpreted as solving a so-called Satisfiability Modulo Theories (SMT). The name indicates that not only a Boolean satisfiability problem (SAT) is solved (as in SAT-based system synthesis), but instead a SAT instance with respect to a given background theory. In the general case, the background theory is given by the set of evaluator functions. In the following, a system synthesis methodology based on SMT-solving will be termed SMT-based System Synthesis (3S).

In the following, the underlying idea of 3S, i.e., an iterative learning approach is formally introduced. Based on this formalization, it is possible to express existing approaches which rely on exhaustive search to find implementations that fulfill nonfunctional constraints as a simple learning scheme. Afterwards, the main contribution of 3S, i.e., early learning based on feasibility checking of partial implementations that allows to significantly decrease runtime by improving the learning due to pruning larger parts of the design space is proposed. The section is concluded by a theoretical investigation of the proposed 3S, proving correctness as well as defining essential requirements for the used evaluator functions.

### 4.1 Learning the Feasible Region

According to the refined problem formulation in Section 3, the set of feasible implementations \( X_f \) can be expressed by its characteristic function

\[
\Psi_f = \Psi_f \land \Psi_N.
\]

In this context, it can be observed that there exists no simple constructive approach to determine \( \Psi_N \), i.e., the characteristic function encoding the implementations that respect all nonfunctional constraints. This holds especially true if nonfunctional constraints are checked by complex calculations or even simulations. Thus, we propose an indirect and iterative construction of \( \Psi_N \) based on so called learning: Given the characteristic function \( \Psi_N \) with \( \Psi_N \lor \Psi_N = 1 \), \( \Psi_f \land \Psi_N = 0 \) encoding all implementations that do not respect the given nonfunctional constraints, we reformulate the calculation of \( \Psi_f \) as follows:

\[
\Psi_f = \Psi_f \land \overline{\Psi_N}.
\]

This reformulation depicts that \( \Psi_N \) may also be constructed by the negated characteristic function of all implementations that are not feasible with respect to nonfunctional constraints \( \overline{C_N} \). This enables the basic idea proposed in this paper that \( \Psi_N \) will be iteratively constructed by learning \( \Psi_N \). This idea can be outlined as follows: Given \( \Psi_N \) in the \( i \)-th step of the iterative learning, the implementation \( x \) that is gathered by the incorporated SAT-solver in this step is checked for feasibility with respect to nonfunctional constraints by the evaluator functions. If the found implementation \( x \) is feasible, the synthesis problem is solved. If \( x \) is infeasible, the encoding \( \vartheta = \sigma(x) \) of \( x \) is added to the characteristic function \( \Psi_N \) and, thus, is excluded from \( i \)-th step. In the following steps (\( > i \)).

---

**Example**

A binding and allocation that is feasible with respect to functional constraints is, e.g., binding all processes \( p_1, \ldots, p_5 \) to resource \( r_1 \), i.e., \( \mu_{i1} = 1, \mu_{i2} = \mu_{i3} = \mu_{i4} = 0 \) with \( i = 1, \ldots, 5 \).

Conceptually, a SAT-solver traverses a so called decision tree by subsequently assigning variables. If an assignment results in a constant evaluation of a given Boolean function \( \Psi \) having the value 0, previous assignments have to be reverted, if possible. If such a backtracking is not possible any more, because all (reasonable) variable assignments have already been tested, the function \( \Psi \) is unsatisfiable and, in our case, no feasible allocation \( A \) and binding \( B \) exists.

A decision tree for the example shown in Figure 1 is shown in Figure 2.\(^4\) Vertices in the decision tree are associated with Boolean variables: Outgoing dashed edges indicate a 0-assignment to the variable associated with the respective vertex. Outgoing solid edges represent a 1-assignment. In a decision tree, each path leading from the root to the 1 terminal node represents a feasible variable assignment and, thus, a feasible allocation and binding. As the SAT-solver performs a depth-first search, the decision tree is not constructed in general. However, in order to illustrate the proposed approach, we will stick to decision trees to represent the search space.

---

**4. SMT-BASED SYSTEM SYNTHESIS**

Beside functional constraints often nonfunctional constraints are imposed on the implementation. Examples for nonfunctional constraints are deadlines, area or power consumption, etc. Although the allocation and binding are both feasible regarding functional constraints, in case a task misses its deadline, the implementation obviously becomes infeasible due to the same resource use intra-resource communication, the binding variables \( \mu \) for the communication processes \( p_2 \) and \( p_4 \) are directly implied by the binding of the computational processes \( p_1, p_3, \) and \( p_5 \), cf. Eq. (4). Thus, for simplicity, this implied variable assignments are not shown in the decision tree, but only the binding variables for the computational processes.

---

\(^4\)Since there exists only one bus and processes bound to the same resource use intra-resource communication, the binding variables \( \mu \) for the communication processes \( p_2 \) and \( p_4 \) are directly implied by the binding of the computational processes \( p_1, p_3, \) and \( p_5 \), cf. Eq. (4). Thus, for simplicity, this implied variable assignments are not shown in the decision tree, but only the binding variables for the computational processes.

---

\(^5\)The implementation already fulfills the functional constraints \( C_f \) since \( \Psi_f \) is available from the beginning.
4.2 Simple Learning

The approach termed simple learning is a straightforward technique that directly results from Eq. (6) and the idea of subsequently learning infeasible implementations. This approach corresponds to the common practice in system synthesis, i.e., an implementation is gathered and checked for feasibility. If it is infeasible, it is excluded from the design space and another implementation is gathered until (1) a feasible implementation is found or (2) no feasible implementation can be found anymore. Although recent improvements allow to restrict the design space to implementations that are feasible with respect to functional and linear constraints, cf. Sec. 2, this scheme is an exhaustive search approach with respect to nonfunctional nonlinear constraints. However, the advantage of the formalization in the work at hand is that excluded implementations due to nonfunctional constraints are symbolically represented which allows for a compact encoding and an effective consideration directly by the incorporated SAT-solver.

Simple learning can formally be outlined as follows: The solving procedure starts with $\Psi_f(0) = 0$:

$$\Psi_f(i) = \Psi_F \land \overline{\Psi_f(i - 1)} = \Psi_F$$

After each feasible variable assignment $\vartheta = \sigma(x)$ encoding a feasible implementation $x$, gathered in the $i$-th SAT-solver run, the evaluations functions are invoked. In case $\vartheta$ is infeasible, $\vartheta$ is added to $\overline{\Psi_f(i)}$ using disjunction. This results in a modified $\Psi_f$ for the next SAT-solver run:

$$\Psi_f(i) = \Psi_F \land (\Psi_f(i) \lor \vartheta)$$

This learning scheme approximates the true $\Psi_f$ over time. Especially in case there exists no feasible implementation of an architecture on a given platform, $\Psi_F \land \overline{\Psi_f(i)}$ will at some point evaluate to 0, such that the absence of a feasible implementation can be proven by the proposed approach.

Example

Assume that the first implementation found by the SAT-solver is the single processor implementation running all processes on the first processor $r_1$. Now, a timing analysis carried out for this implementation reports a deadline violation. This timing analysis is the background theory of the given example. As the single processor implementation is represented by $\vartheta_1 = \mu_11 \land \ldots \land \mu_51 \land \mu_52 \land \mu_52 \land \mu_53$, the formula $\Psi_N$ that learns the infeasibility of this implementation is:

$$\Psi_f(1) = \Psi_f(0) \lor \vartheta_1$$

Considering again the decision tree shown in Figure 2, one can see that eight satisfiable variable assignments exist (paths leading to the 1 terminal node). In the case that it is not possible to feasibly implement the application on the given platform due to nonfunctional constraints, also the nonfunctional constraint checking is performed eight times. In this case, eight infeasible implementations $\vartheta_1, \ldots, \vartheta_8$ will be learned leading to $\Psi_f \land \overline{\Psi_f(8)} = 0$, which proves the infeasibility of this example due to nonfunctional nonlinear timing-constraints.

The obvious drawback of this simple learning technique is that in the worst-case (no feasible implementation is available), the evaluator functions are called $n$-times with $n$ being the number of implementations that are feasible with respect to the functional constraints, cf. Figure 3a. Not only the sheer number of feasibility checks, but also the length of the clauses that result from adding the encoding $\vartheta = \sigma(x)$ of each infeasible implementation $x$ may become a problem for an efficient solving of the resulting function $\Psi_f$. As a remedy, the following section introduces an early learning scheme.

4.3 Early Learning

As performance evaluation and feasibility checking are by far the most expensive operations in system synthesis, reducing the number of evaluation runs would significantly contribute to the overall performance of a system synthesis methodology. As a consequence, feasibility checking should be performed on partial allocations $A \subseteq A$ and bindings $B \subseteq B$. Thus, we propose early learning based on feasibility checking of partial implementations which is enabled by the introduced formalization of learning the infeasible region and introduced in the following.

Example

Looking again at the decision tree shown in Figure 2, it is assumed WLOG that the SAT-solver traverses the decision tree from left to right. After binding processes $p_1$ and $p_3$ to resource $r_1$, the binding of the communication process $p_2$ will be implied. The partial binding $\tilde{B}$ could be represented as $\vartheta_1 = \mu_11 \land \mu_23 \land \mu_31$. Assuming performance analysis detects an overload of resource $r_1$ (which might be the cause for the deadline violation in the above single processor implementation) two feasible allocations and bindings could be ruled out by a single nonfunctional constraint check, cf. Figure 4a. This is because of the fact that any implementation binding $p_1$ and $p_3$ to resource $r_1$ will violate the timing constraints independent of the actual binding of process $p_3$.

Hence, learning the clause $\vartheta_1$ does not only remove a single implementation from the decision tree, but instead an entire subtree is pruned. Going on in the platform-based synthesis, a next partial allocation and binding is represented by $\vartheta_2 = \mu_11 \land \mu_23 \land \mu_32$. Thus, processes
Figure 4: Five steps of design space pruning using SMT-based system synthesis. In this small example, six evaluations including one ineffective check (*) would be enough to check the whole design space, while former techniques need to check all eight possible paths.

$p_1$ and $p_2$ are bound to resources $r_1$ and $r_2$, respectively, while $p_3$ is bound to the bus $r_3$. Assuming now that $p_2$ causes a too high load on the bus, again two feasible allocations/bindings could be removed from the design space with a single nonfunctional constraint check, cf. Figure 4b. In summary, the SAT-solver learned that binding process $p_1$ to resource $r_1$ leads to ineffective implementations. As a result, process $p_1$ will be bound to $r_2$ in the next step. Still traversing the decision tree in Figure 4 from left to right, process $p_3$ will be bound onto resource $r_1$. As an implication, $p_2$ again will be bound to the bus $r_3$. The following check of nonfunctional constraints will detect the same overload on the bus as before, and providing this information to the SAT-solver further prunes the search space, cf. Figure 4c.

Finally, after binding both processes $p_1$ and $p_3$ to $r_2$, the timing analysis will not report any timing constraint violation, cf. Figure 4g, such that this call to the timing analysis can be considered ineffective. Thus, process $p_2$ has to be bound before a final timing analysis could be done. This results in two additional tests ($\mu_{51} = 1, \mu_{52} = 0$ and $\mu_{51} = 0, \mu_{52} = 1$, cf. Figure 4d resp. e), before concluding that the given application cannot be feasibly implemented on the given platform.

From this small example, two important observations for 3S can be made: (1) Doing early learning by feasibility checking of partial allocations/bindings leads to pruning bigger areas in the search space, cf. Figure 3b. Consequently, this approach could lead to a smaller number of nonfunctional constraints checks. In the running example, the simple learning required eight checks of nonfunctional constraints whereas early learning did only require six checks. (2) As the nonfunctional constraint check is performed on a partial allocation/binding, the required runtime of this check will probably be shorter than running a test on a complete allocation/binding. The reason is that the majority of evaluation methods for embedded systems expose that runtime scales in the number of allocated and/or bound objects. As a result, the overall runtime of the synthesis will further be shortened. However, one has to be careful relying on the first item. In particular, running too many intermediate tests on partial allocations and bindings may increase the overall number of nonfunctional constraint checks.

The proposed idea can be formalized as follows: With 3S, not only complete but also partial implementations are added to $\Psi_N$. Let $\bar{x} = (A, B, T)$ be a partial implementation with respect to $x = (A, B, T)$, written as $\bar{x} \subseteq x$. It holds:

$$\bar{x} \subseteq x \iff \sigma ^{-1}(\bar{\vartheta}) = \bar{x}$$

with $\bar{\vartheta} \subseteq \vartheta$ indicating that $\bar{\vartheta}$ is a partial variable assignment with respect to $\vartheta$, i.e.,

$$\bar{\vartheta} \subseteq \vartheta \iff \bar{\vartheta} \lor \vartheta = \vartheta$$

and $\sigma ^{-1}$ being the inverse function with respect to $\sigma$

$$\forall x \in X : x = \sigma ^{-1}(\sigma(x)).$$

Given an evaluation function$^6$ $g \in G$ with $g : 2^X \to \{0,1\}$ evaluates to 0 if the implementation $x$ is infeasible. With the proposed idea of an feasibility checking, Eq. (7) can be rewritten such that also partial variable assignments $\bar{\vartheta}$ can be learned:

$$\Psi^{(i)}_N = \Psi^{(i-1)}_N \lor (g(\sigma ^{-1}(\bar{\vartheta})) \land \bar{\vartheta})$$

$$\Psi^{(i)}_F = \Psi_F \land \Psi^{(i)}_N$$

However, not all evaluation and feasibility checking methods can be reasonable applied to partial implementations. In the following section, the correctness of 3S will be shown and the requirements for using feasibility checking methods in the 3S approach are discussed. A fundamental result will be that the methods have to be monotonic in order to avoid false negative results, i.e., pruning feasible implementations from the design space.

4.4 Theoretical Investigations

In this section, the proof of the correctness of the proposed SMT-based system synthesis is given. In particular, it is shown that 3S finds a feasible implementation if present or proves the absence of any feasible implementation, respectively. Moreover, requirements for the correct integration of external analysis tools as feasibility checkers are derived. Particularly, it is proven that the external analysis has to be monotonic. Moreover, we introduce so called barrier functions that only allow the invocation of the analysis in case monotonicity is ensured so to enable the use of analysis approaches that are not monotonic when applied to any partial implementation.

For the following investigations, two axioms are given:

Axiom 1. Given a characteristic function $\Psi$, the SAT-solver finds a variable assignment such that $\Psi$ evaluates to 1 if present or proves the absence of such an assignment, cf. [4].

Axiom 2. The characteristic function $\Psi_F$ is a correct encoding of $X_F$, cf. [10].

Lemma 1. The proposed learning scheme terminates.

Proof by Cases. Based on Axiom 1, for any iterative learning step $i$ exactly one of the following cases holds: (1) A variable assignment $\vartheta$ that fulfills $\Psi^{(i)}_F$ is found in the $i$-th step. If the implementation $x = \sigma ^{-1}(\vartheta)$ under investigation is feasible, i.e., it fulfills the nonfunctional constraints such

$^6$WLOG a single evaluation function in the set of evaluation functions $G$ is assumed throughout the paper.
that \( g(x) = 1 \), then the procedure terminates and returns \( x \). Otherwise, \( x \) is infeasible, i.e., \( g(x) = 0 \), and learned, cf. Eq. (11a). Thus, \( \vartheta \) is permanently removed from \( \Psi_f^{(n)} \) with \( n \geq (i+1) \) and will not be found by the SAT-solver again, cf. Axiom 1. (2) No variable assignment \( \vartheta \) that fulfills \( \Psi_f^{(i)} \) can be found and, thus, the absence of a feasible implementation is proven, cf. Axiom 1, i.e., \( \Psi_f \land \overline{\Psi_f} = \emptyset \) and the procedure terminates.

Given Lemma 1 that proves the termination of the proposed approach and Axiom 2 that ensures correctness of the encoding of the functional constraints \( \Psi_f \), it needs to be proven that the proposed learning scheme is correct.

**Definition 1.** Learning based on the scheme given by Eq. (6) is correct if (1) no infeasible implementation \( x \notin X_f \) is regarded as feasible and (2) no feasible implementation \( x \in X_f \) is pruned from the search space.

Definition 1(1) is ensured by Eq. (7) and (11a) and the fact that each implementation is checked for feasibility using the evaluation function \( g \) before being returned. For the simple learning, ensuring Definition 1(2) is trivial, since each implementation \( x \) under investigation is explicitly checked for feasibility by applying \( g(x) \) before being pruned from the design space. However, 3S allows to learn partial variable assignments. In the following, it is proven that, in order to ensure Definition 1(2), the used evaluation function \( g \) has to be monotonic.

**Lemma 2.** Adding a partial variable assignment \( \tilde{\vartheta} \) to the set of infeasible implementations \( \Psi_f^{(i)} \) is only valid if all corresponding implementations \( x \in \{ \sigma(\vartheta) | \tilde{\vartheta} \subseteq \vartheta \} \) are infeasible as well.

**Proof by Contradiction.** Given a partial variable assignment \( \tilde{\vartheta} \) and an implementation \( x \in X_f \) that is feasible, i.e., \( g(x) = 1 \). Particularly, \( x = \sigma^{-1}(\vartheta) \) is a partial implementation of \( x \), i.e., \( \tilde{\vartheta} \subseteq \vartheta \) with \( \vartheta = \sigma(x) \). Assume that the partial implementation \( x = \sigma^{-1}(\vartheta) \) is evaluated to be infeasible, i.e., \( g(x) = 0 \) in iteration \( i \). Thus, \( \vartheta \) is added to the set of infeasible implementations by disjunction to \( \Psi_f^{(i)} \).\( \Psi_f \), respectively. In particular, with \( \forall n \geq i : \vartheta \land \Psi_f^{(n)} = 0 \), this implies \( g(x) = 0 \) which contradicts the Definition 1(2) since an in fact feasible implementation \( x \) is pruned from the design space.

Since Lemma 2 has to be fulfilled in order to perform a correct learning, it can be concluded that the evaluation function \( g \) has to be monotonic. An evaluation function \( g : 2^X \rightarrow \{0, 1\} \) is called monotonic, if

\[
\forall \tilde{\vartheta}, \vartheta \text{ with } \tilde{\vartheta} \subseteq \vartheta : g(\sigma^{-1}(\tilde{\vartheta})) \geq g(\sigma^{-1}(\vartheta)).
\]  

(12)

**Lemma 3.** If the evaluation function \( g \) is monotonic, no feasible implementation \( x \in X_f \) is pruned from the search space, cf. Definition 1(2).

\[
\forall i : \exists ! x \in X_f : \sigma(x) \land \Psi_f^{(i)} = 0.
\]  

(13)

**Proof by Induction.** Given Eq. (13), it especially holds that for each feasible implementation \( x \):

\[
\sigma(x) \land \Psi_f^{(i)} = \sigma(x).
\]  

Let \( n = 0: \Psi_f^{(0)} = \Psi_f \) be correct, cf. Axiom 2. Let \( n = i - 1 \) be correct.

\[
r(\sigma^{-1}(\tilde{\vartheta}))
\]

Figure 5: The reliability of a partial system against the number of decided variables. The vertical lines show, where \( b(\tilde{\vartheta}) = 1 \) to get a monotonic approximation (horizontal curve).

- If the partial implementation \( \tilde{x} \) is feasible, i.e. \( g(\tilde{x}) = 1 \), then according to Eq. (11a)

\[
\Psi_f^{(i)} = \Psi_f \land \overline{\Psi_f} \lor (0 \land \tilde{\vartheta}) = \Psi_f^{(i-1)}.
\]

Thus, \( \Psi_f^{(i)} \) is correct.

- If the partial implementation \( \tilde{x} \) is infeasible, i.e. \( g(x) = 0 \), then

\[
\Psi_f^{(i)} = \Psi_f \land \overline{\Psi_f} \lor \tilde{\vartheta} = \Psi_f^{(i-1)} \lor \tilde{\vartheta}
\]

(15)

Assume there now exists a feasible (complete) implementation \( x \in X_f, g(x) = 1 \) with \( \vartheta = \sigma(x) \) that is pruned from \( \Psi_f^{(i)} \), i.e.

\[
\vartheta \land \Psi_f^{(i)} = 0
\]

\[
\Rightarrow \quad \vartheta \land \overline{\Psi_f^{(i-1)}} = 0
\]

According to the assumption \( n = i - 1 \) is correct and Eq. (14) it holds that \( \vartheta \) is a partial implementation of \( \vartheta \):

\[
\vartheta \land \overline{\vartheta} = 0 \quad \Rightarrow \quad \sigma^{-1}(\vartheta) \subset \sigma^{-1}(\tilde{\vartheta})
\]

Since \( \vartheta \) is infeasible, i.e., \( g(\sigma^{-1}(\tilde{\vartheta})) = 0 \) and \( \vartheta \) is feasible, \( g(\sigma^{-1}(\tilde{\vartheta})) = 1 \), it follows that

\[
g(\sigma^{-1}(\tilde{\vartheta})) < g(\sigma^{-1}(\vartheta))
\]

(16)

which contradicts Eq. (12). Thus, a feasible implementation can only be pruned if \( g \) is not monotonic.

Many real-world nonfunctional constraints are not monotonic according to Lemma 3. For example, consider a reliability constraint: Determining the reliability of the system after each variable decision results in a non-monotonic function, cf. Figure 5. This is due to the fact that, e.g., additional redundancy by allocating more resources may be implied by variables that are set later in the decision process. However, in case all variables that influence the reliability of one application are set, the upper bound for the reliability of that application can be calculated. By doing so for each application, the reliability calculation for the overall system becomes monotonic, cf. Figure 5. It, thus, can be observed.
that evaluation functions can be made monotonic\(^7\) by allowing evaluations only at certain levels in the decision tree that we call barriers, cf. Figure 6. Therefore, a so called barrier function \(b_g : 2^\mathbb{X} \rightarrow \{0,1\}\) is defined that evaluates to 1, iff monotonicity of the corresponding evaluation function \(g\) is ensured. Thus, Eq. (11a) can be amended by \(b_g\) to
\[
\Psi_N^{(i)} = \Psi_N^{(i-1)} \lor (b_g(\tilde{\vartheta}) \land g(\sigma^{-1}(\vartheta)) \land \tilde{\vartheta)) \tag{17}
\]
Besides the ability to ensure monotonicity of the evaluation functions, \(b\) can be used to reduce the number of evaluations of the background theory, which can improve the efficiency of the approach as the evaluation overhead can be reduced.

At this point, it should be mentioned that the used variable order when solving \(\Psi_f\) may strongly influence the performance of the SAT-solver which is an extensively studied topic, see for example \cite{11}. In the proposed 3S approach, the used variable order may also heavily influences when the barrier function \(b_g\) evaluates to 1. Of course, an evident goal could be to have \(b_g\) evaluating to 1 as early in the decision process as possible such that larger parts of the design space can be pruned. Although a detailed investigation of the influence of different variable orders and available heuristics is out of the scope of this paper, a coarse investigation of the influence of the used variable order on the 3S approach is carried out by applying a simple heuristic: It clusters variables that are responsible for the allocation and binding of a single application given by a directed application graph. The idea is that several feasibility checks may only be correct (\(b_g = 1\)) if an application under investigation is completely bound to the platform. A comparison of this heuristic and random variable ordering is presented and discussed in Section 5.3.

5. EXPERIMENTAL RESULTS

In this section, the results of applying the methodology proposed in the work at hand to the design of an automotive ECU network are presented. After the introduction of the used case study, the proposed early learning is compared to the simple learning scheme. The section is concluded by an investigation of the used variable order in 3S.

\(^7\)In case no monotonicity can be ensured, there exists at least one barrier after all variables have been set which corresponds to simple learning.

Table 1: Deadlines and analyzed delays of the individual applications and the reference implementation (app\(_1\) and app\(_2\)).

<table>
<thead>
<tr>
<th>Application</th>
<th>Deadline [ms]</th>
<th>Delay [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{app}_1 )</td>
<td>28.5</td>
<td>19.55</td>
</tr>
<tr>
<td>( \text{app}_2 )</td>
<td>27.55</td>
<td>22.99</td>
</tr>
<tr>
<td>( \text{app}_3 )</td>
<td>9.6</td>
<td>9.35</td>
</tr>
</tbody>
</table>

5.1 Case Study

As a case study, a typical automotive subnet consisting of 15 Electronic Control Units (ECU), two CAN buses, and one FlexRay bus, in particular its so called dynamic segment is used. The buses are connected by a gateway. Distributed on this network, two applications app\(_1\) and app\(_2\) are running. The task is to find a feasible rearrangement of the tasks such that an additional application app\(_3\) from the body domain can be executed on the given platform. Altogether, the resulting system consists of 27 tasks and 24 periodic messages. We follow the encoding scheme of \cite{7}, i.e., the mapping of tasks, the routing of messages, as well as the allocation of resources can be modified during the optimization. The scheduling \(T\) is given by priority-based scheduling on each ECU, the CAN buses, and the dynamic segment of the FlexRay bus as well as by priorities for each message and task. In particular, the priorities for app\(_1\) and app\(_2\) are taken from the reference implementation and are, for safety reasons, higher than the designer-specified priorities for the infotainment application app\(_3\).

The timing constraints are given as end-to-end deadlines for each application as listed in Table 1. Note that each application alone can be feasibly implemented on the given platform, cf. Table 1. Moreover, preprocessing revealed that any combination of two of the three applications can be feasibly implemented on the given platform. The end-to-end deadlines of the combination of app\(_1\) and app\(_2\), i.e., the reference implementation, is depicted in Table 1 as well.

For Modular Performance Analysis (MPA), Real-Time Calculus (RTC) \cite{9, 22} is used. As the arrival and service curves are monotonic and Min-Max-Algebra is used \cite{23}, each additional decision leads to higher delays. Therefore, \(\forall \vartheta \in \Theta : b_{\text{MPA}}(\vartheta) = 1\) is correct. However, since only the end-to-end latency of each application is constrained, three barriers are used with \(b_{\text{MPA}}(\vartheta) = 1\) whenever a complete application is mapped to the platform. The experiments were carried out on one core of an Intel Core2 Quad with 2.66 GHz having 2 GByte RAM available. For each investigated problem instance, 10 independent synthesis runs have been carried out.

5.2 Simple Learning vs. Early Learning

With the deadlines as specified in Table 1, the combination of the three applications is not possible, i.e., there exists no feasible implementation on the given platform.\(^8\) Because of the complexity of this real-world example, this fact could not be proven using the simple learning strategy. The timing analysis could be skipped to save time by simply pruning every found implementation from the design space by adding the according clause to \(\Psi_N\). After 1 059 223 learned implementations, the given memory bound of 2 GByte was exceeded, thus, it was not even possible to traverse the whole search space with a current SAT solver. Even if the available memory is increased to 6 GByte, the memory is still not sufficient and the synthesis fails after 3 928 478 learned implementations. As the timing analysis of one implementation lasts for about 4 s on average, the overall runtime would

\(^8\)This was proven by applying early learning with 3S first.
have been about 180 days, terminating without proving infeasibility.

On the other hand, 3S succeeds in showing that no feasible implementation with the given deadlines exists. In this case, the timing analysis was activated such that also the timing behavior of the proposed approach in combination with the used Modular Performance Analysis can be investigated. On average, after running for 21 hours 48 minutes and 29 seconds a contradiction shows that there is no feasible implementation regarding the given constraints and the learned clauses, i.e., $X_f = \emptyset$, cf. Table 2.

Up to this point, the timing of about 75,000 partial implementations has been evaluated on average. Most of the time, 20 hours 35 minutes and 51 seconds, were consumed by the timing analysis with MPA, an average of 988.4 ms per evaluation. Having a look at a particular synthesis run, i.e., run1, the timing of 47,670 partial implementations has been evaluated out of which 45,291 were infeasible. Thus, $\approx 95\%$ of the feasibility checks revealed infeasible implementations which shows the effectiveness of the early learning and an overhead of only $\approx 5\%$ uneffective checks. The presented case study indicates that the proposed 3S approach using early learning is by far superior to the simple learning in case there exists no feasible implementation. In particular, even after a runtime that is two orders of magnitude higher, the simple learning finally fails to prove infeasibility because the learned $\Psi_f$ outgrows given memory.

To investigate the overhead in case there exist feasible implementations, the given deadlines are step-wise relaxed. If the deadline of app1 is increased by $0.625\%$ to 9.66 ms, a feasible implementation is possible and found with the presented technique. The marginal difference of $0.625\%$ regarding the deadline of only one application between absence and presence of a feasible implementation highlights the importance of an automatic system synthesis. By using former techniques to find a feasible implementation for the extended system, it is again only possible to restrict $\Psi_f$ after the evaluation of a complete system. After the evaluation of more than 200,000 different implementations and a run time of 2 hours 47 minutes and 22 seconds no feasible implementation was found yet. Moreover, even the best implementations found till this point still have two paths which miss their deadlines. If the deadlines are relaxed further (app1 and app2 by $57.9\%$, app3 by $57.5\%$), the runtime decreases as the number of feasible implementations increases, cf. Table 3. But even with much weaker timing constraints, the SMT-based system synthesis is still slightly faster on average than the simple learning strategy which now also finds feasible implementations, cf. Table 4. This is due to the fact that although 3S introduces overhead due to ineffective feasibility checks, the partial evaluations require less runtime than the evaluation of complete implementations. If the deadline constraints would be further relaxed until only some implementations do not meet the timing constraints, the overhead of SMT-based system synthesis would lead to longer runtimes due to the ineffective evaluation of the partial implementations.

### Table 2: Synthesis runs with strict deadlines $d_{app1} = 28.5$ ms, $d_{app2} = 27.55$ ms, $d_{app3} = 9.6$ ms that lead to $\mu = \emptyset$, $\forall i$. The second column shows the overall runtime of the synthesis run. The third column shows the number of partial evaluations till the resulting contradiction, which is also the number of MPA runs. The forth column shows the average time per real time analysis.

<table>
<thead>
<tr>
<th>runtime [s]</th>
<th>evaluations</th>
<th>$\tau_{MPA}$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>run0</td>
<td>8 480</td>
<td>47 670</td>
</tr>
<tr>
<td>run1</td>
<td>4 143</td>
<td>37 518</td>
</tr>
<tr>
<td>run2</td>
<td>200 612</td>
<td>142 792</td>
</tr>
<tr>
<td>run3</td>
<td>207 103</td>
<td>100 420</td>
</tr>
<tr>
<td>run4</td>
<td>4 146</td>
<td>31 828</td>
</tr>
<tr>
<td>run5</td>
<td>30 227</td>
<td>114 717</td>
</tr>
<tr>
<td>run6</td>
<td>40 930</td>
<td>81 571</td>
</tr>
<tr>
<td>run7</td>
<td>12 543</td>
<td>57 217</td>
</tr>
<tr>
<td>run8</td>
<td>148 881</td>
<td>29 611</td>
</tr>
<tr>
<td>run9</td>
<td>128 024</td>
<td>106 872</td>
</tr>
<tr>
<td>average</td>
<td>78 509</td>
<td>75 021.6</td>
</tr>
</tbody>
</table>

### Table 3: SMT-based system synthesis with relaxed timing constraints. Deadlines are now $d_{app1} = 45$ ms, $d_{app2} = 43.5$ ms, $d_{app3} = 18$ ms.

<table>
<thead>
<tr>
<th>runtime [s]</th>
<th>evaluations</th>
<th>$\tau_{MPA}$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>run0</td>
<td>12</td>
<td>33</td>
</tr>
<tr>
<td>run1</td>
<td>10</td>
<td>27</td>
</tr>
<tr>
<td>run2</td>
<td>26</td>
<td>49</td>
</tr>
<tr>
<td>run3</td>
<td>62</td>
<td>32</td>
</tr>
<tr>
<td>run4</td>
<td>26</td>
<td>100</td>
</tr>
<tr>
<td>run5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>run6</td>
<td>22</td>
<td>41</td>
</tr>
<tr>
<td>run7</td>
<td>356</td>
<td>342</td>
</tr>
<tr>
<td>run8</td>
<td>402</td>
<td>162</td>
</tr>
<tr>
<td>run9</td>
<td>23</td>
<td>122</td>
</tr>
<tr>
<td>average</td>
<td>94.8</td>
<td>92.1</td>
</tr>
</tbody>
</table>

### Table 4: Plain SAT-based system synthesis with relaxed timing constraints $d_{app1} = 45$ ms, $d_{app2} = 43.5$ ms, $d_{app3} = 18$ ms.

<table>
<thead>
<tr>
<th>runtime [s]</th>
<th>evaluations</th>
<th>$\tau_{MPA}$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>run0</td>
<td>149</td>
<td>20</td>
</tr>
<tr>
<td>run1</td>
<td>123</td>
<td>23</td>
</tr>
<tr>
<td>run2</td>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>run3</td>
<td>179</td>
<td>27</td>
</tr>
<tr>
<td>run4</td>
<td>174</td>
<td>94</td>
</tr>
<tr>
<td>run5</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>run6</td>
<td>38</td>
<td>21</td>
</tr>
<tr>
<td>run7</td>
<td>185</td>
<td>18</td>
</tr>
<tr>
<td>run8</td>
<td>36</td>
<td>13</td>
</tr>
<tr>
<td>run9</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>average</td>
<td>95.6</td>
<td>24.1</td>
</tr>
</tbody>
</table>

### 5.3 Variable Order

To show the influence of the used variable order, Tables 2-4 explicitly show the results for 10 synthesis runs with different variable orderings. The huge variations in runtime that can be observed is the result of (1) varying runtimes of the used Modular Performance Analysis and (2) a varying number of partial evaluations needed. The former is inherent to the used MPA approach which can get very complex if the timings of different parts of the system influence each other. However, the former and the latter cause of variation are not completely independent. The number of evaluations needed depends on the variable order that again also influences which parts of the system are allocated and bound first and, thus, are analyzed by MPA. The influence of the variable order is especially demonstrative in Tables 3-4 where the approach terminates as soon as the first feasible implementation is found. However, variation is also given in Table 2 where there exists no feasible implementation.

As suggested in Section 4.4, a simple heuristic for the variable order is applied in this problem that aims to cluster the variables responsible for the binding of a single application. Note that an ordering of the clusters only results in $3! = 6$ cluster orders, but the order of the variables within each cluster is still randomly generated. The results for the case study with strict deadlines, cf. Tables 1 and 2, achieved by using the heuristic are depicted in Table 5. It can be
Table 5: Using an optimized barrier function the efficiency of the approach can be increased.

<table>
<thead>
<tr>
<th>runtime [s]</th>
<th>evaluations</th>
<th>\text{MPA} [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>run0</td>
<td>19 694</td>
<td>27 341</td>
</tr>
<tr>
<td>run1</td>
<td>22 656</td>
<td>29 071</td>
</tr>
<tr>
<td>run2</td>
<td>9 127</td>
<td>24 842</td>
</tr>
<tr>
<td>run3</td>
<td>15 152</td>
<td>28 723</td>
</tr>
<tr>
<td>run4</td>
<td>11 855</td>
<td>37 025</td>
</tr>
<tr>
<td>run5</td>
<td>13 899</td>
<td>28 105</td>
</tr>
<tr>
<td>run6</td>
<td>15 203</td>
<td>25 694</td>
</tr>
<tr>
<td>run7</td>
<td>15 064</td>
<td>30 050</td>
</tr>
<tr>
<td>run8</td>
<td>18 592</td>
<td>28 512</td>
</tr>
<tr>
<td>run9</td>
<td>12 499</td>
<td>30 623</td>
</tr>
<tr>
<td>average</td>
<td>15 383</td>
<td>28 998.6</td>
</tr>
</tbody>
</table>

observed that the average number of evaluations until the result is computed decreases strongly and, hence, also the overall runtime. The average runtime could be decreased by $\approx 80.4\%$ which indicates that more sophisticated and maybe problem dependent approaches for determining a good variable order may strongly enhance the efficiency of the proposed 3S approach. In particular, the average time to prove infeasibility of the case study decreased to 4 hours 16 minutes and 23 seconds with a much lower variation compared to the random variable order. The lowest runtime using the proposed heuristic is about 2.2 times higher than the lowest runtime observed with a random variable order. On the other hand, the highest runtime using the proposed heuristic is about 9.1 times lower than the highest runtime observed with a random variable order. This again emphasizes the need for more detailed investigations of the used variable order in the future.

6. CONCLUSIONS

In this work, an approach based on indirect Satisfiability Modulo Theorems (SMT) solving is presented that efficiently learns the set of implementations that are infeasible with respect to nonfunctional nonlinear constraints. In particular, the proposed approach termed SMT-based System Synthesis (3S) contributes: (1) An early learning approach based on feasibility checking of partial implementations that allows to speed up the search for feasible implementations by pruning whole sets of implementations that are infeasible due to nonfunctional nonlinear constraints from the search space. (2) For the first time, the ability to efficiently prove whether there exists a feasible implementation of a set of applications on a given platform by incorporating symbolic techniques instead of using costly exhaustive search approaches. (3) The seamless integration of external analysis approaches, e.g., for performance, real-time, or power analysis for checking nonfunctional constraints is shown. Thus, existing system synthesis approaches that incorporate several external analysis tools may also benefit from the proposed formal technique.

Using a case study from the automotive domain where Modular Performance Analysis is used to check end-to-end latencies, the effectiveness of the proposed approach is investigated. It can be observed that 3S in combination with the proposed early learning outperforms a simple learning approach that is comparable to existing synthesis approaches on average. In particular, an average runtime of $\approx 4.25$ hours for 3S to prove that there exists no feasible implementation of the applications on the given platform for the original deadlines has been observed. The simple learning scheme would have finished after $\approx 180$ days by failing to prove infeasibility since the data structure of the SAT-solver outgrew the available memory. Moreover, the deadlines of the case study have been subsequently relaxed such that there exist feasible implementations. In these cases, the overhead introduced by the early learning is still competitive with the simple learning strategy since the faster evaluation of partial implementations compensates ineffective evaluations. Overall, it can be expected that 3S outperforms existing system synthesis approaches in the presence of stringent linear and, as a unique feature, nonlinear constraints that become more and more important in the design of embedded systems.

7. REFERENCES