SMT-Based Optimization for Synchronous Programs

Yu Bai, Jens Brandt and Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de/

Abstract
In this paper, we present several optimization techniques to improve the runtime and size of the code generated from synchronous programs. These optimizations work on extended finite state machines (EFSMs) that can be used as intermediate representation for any synchronous system. Our optimizations consist of two phases: First, local optimization guides the EFSM generation and considers the states and edges separately. Second, global optimization is based on a dataflow analysis of the entire EFSM. For both phases, we employ an SMT (Satisfiability Modulo Theories) solver to verify the individual optimization steps. Our experiments show the potential of the presented optimizations: optimized programs generally have a smaller size and a better run-time performance.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—compilers and code generation; D.3.2 [Programming Languages]: Language Classifications—concurrent, distributed, and parallel languages

General Terms Algorithms, Languages

Keywords Synchronous Languages, Code Generation, Satisfiability-Modulo-Theory, Extended Finite State Machines

Acknowledgment
We would like to thank Niklas Holsti from Tidorum Ltd. for providing useful comments and discussions about the experiments with Bound-T.

1. Introduction
Compared to other application areas, software development for embedded systems is much more challenging [23, 35], since many additional aspects have to be considered. Besides many non-functional constraints like robustness with respect to the physical environment, energy efficiency, and the lack of powerful computational resources, the correctness of embedded systems is a major concern. This is not only the case for embedded systems that are used in safety-critical applications, but also for those that are sold in large quantities where replacements would endanger the economic success of the products.

Hence, verification and optimization play an important role in the development of embedded systems. While verification aims at proving that a system meets a given specification, optimization aims to improve the efficiency of a given system. These two tasks are often separated in different phases of a design flow: after modeling a system in a first phase, verification of desired properties is done in a second phase before the model is translated and optimized to the final implementation. However, completely separating these design phases is not recommended, since the information obtained in the different phases could be used in other phases as well [6, 19, 32]. Thus, sharing concepts between different design phases or even integrating them, may reveal new possibilities in system design.

This paper is in the spirit of these ideas. It builds upon two pillars: (1) For modeling and verification of the embedded system, we use synchronous programs that have many advantages, in particular, a formally defined semantics with a deterministic concurrency, which is a necessary requirement for verification and static analysis. Moreover, the absence of dynamic data structures allows a much better static analysis of the runtime requirements. Thus, synchronous programs provide a good basis for our desired combination of verification and optimization. (2) For the verification part, we use a SMT (satisfiability-modulo-theory) prover [3, 11] as a backend. SMT solvers can efficiently check the satisfiability of first-order formulas over so-called background theories.

The contribution of this paper consists in an optimization technique that employs verification for the optimization of synchronous languages. This is not only done for verification of a performed optimization step. In addition, we also generate verification goals during the compilation and optimization process, and hand them over to a tightly coupled SMT-solver. The compilation and optimization process continues according to the decisions made by the SMT solver. We will demonstrate that we can this way achieve a significant improvement of the run-time performance of the generated code. Our code optimization is not based on a particular source language. Instead, it is based on a general intermediate code (synchronous guarded actions) so that our optimization techniques can be applied to several source languages and are also independent of the later on chosen target architecture.

Dataflow analysis has been used for code optimization in compilers for decades. Early work has been presented in [1, 17, 18, 27] and considers different kinds of dataflow analyses like the computation of live variables, busy variables, available expressions (to detect shared expressions), used-def chains and many more. However, these analyses cannot be applied for synchronous programs due to the different underlying model of computation and its different definition of equivalent computations. In particular, actions do not immediately update variables, since changes of the variable’s values are synchronously done at the level of macro steps.
Hence, even though static dataflow analysis is a well-established tool for classic code optimization, it has not yet been widely used for the compilers of synchronous languages. Research over the last two decades has primarily tackled semantical problems (namely causality [4, 30] and schizophrenia [31, 34]), which make the compilation of synchronous programs not at all straightforward.

Finally, various compilers have been developed [5, 7, 26] based on different code generation schemes. Static analysis was only used in [33] to compute conditions for instantaneous execution of a synchronous program that is done similarly by the control flow predicates in [28]. In [8], a static dataflow analysis based on fixpoint computations similar to the classic dataflow analyses was developed, which has a different goal than the techniques presented in this paper: While [8] considers the identification of so-called passive code, this paper determines dead code.

The rest of this paper is structured as follows: Section 2 first considers the foundations of our optimizations. It describes the starting point of our optimization, namely synchronous guarded actions, and gives a brief introduction to SMT. Section 3 considers extended finite state machines (EFSMs) and their generation. Section 4 presents our local optimization strategies, while Section 5 considers the global ones. Section 6 presents experimental results, before we conclude with a short summary in Section 7.

## 2. Foundations

### 2.1 Synchronous Guarded Actions

The optimization methods of the next section do not consider the program at the source code level. Instead, they are based on our Averest Intermediate Format (AIF), which mainly consists of a set of synchronous guarded actions. In general, it is possible to translate any synchronous program, e.g. an Esterel, Lustre or Quartz program, to a set of equivalent guarded actions. The interested reader is referred to related publications such as [7, 29].

Synchronous guarded actions are designed in the spirit of guarded commands [9, 12, 20] which are a well-established formalism for the description of concurrent systems. However, note that in contrast to many other applications where guarded actions are used, the guarded actions considered here follow the synchronous model of computation (MoC) similar to Greenstreet’s synchronized transitions [15], which postulates that a system is executed in discrete reaction steps (macro steps). In each step, the system reads all inputs, performs finitely many computation steps (micro steps) and finally produces the outputs in the same macro step. Since all micro steps executed in a macro step refer to the same variable environment, variables are not modified by single micro steps (actions) of a macro step. Instead, the computations follow data dependencies.

This leads to the programmer’s view that the execution of micro step actions does not take time at all while every macro step of a synchronous program requires the same amount of logical time.

Thus, the starting point of our analysis is a system described by a set of synchronous guarded actions of the form \((\gamma \Rightarrow A)\) defined over a set of variables \(V\). The Boolean condition \(\gamma\) is called the guard and \(A\) is called the action. In this paper, actions are restricted to the assignments of the source language, i.e. the guarded actions have either the form \((\gamma \Rightarrow x \leftarrow \tau)\) (for an immediate assignment) or \((\gamma \Rightarrow \text{next}(x) = \tau)\) (for a delayed assignment). Both kinds of assignments evaluate the right-hand side expression \(\tau\) in the current macro step. Immediate assignments \(x = \tau\) write the obtained value of \(\tau\) immediately to the variable \(x\), whereas delayed ones \(\text{next}(x) = \tau\) write the value in the following macro step.

In addition to the set of guarded actions, there are implicit assignments due to the semantics of the program: The so-called reaction to absence defines the value of a variable if no action determines its value in the current macro step. For a variable \(y\), this is the case iff the guards of all immediate assignments to \(y\) are false in the current step and the guards of all delayed assignments to \(y\) have also been false in the preceding step. In this case, the reaction to absence sets the value of the variable according to its storage mode: Event variables \(\text{EventVars}(V) \subseteq V\) are reset to their default values (like wires in hardware circuits) by the reaction to absence, while \(\text{memorized}\) variables \(\text{MemVars}(V) \subseteq V\) store their previous values (like registers in hardware circuits).

The left-hand side part of Figure 1 gives an example, which consists of two threads. The first one implements a simple arithmetic component that can compute two different functions \(t_1\) and \(t_2\) depending on the Boolean flag \(op\). The second thread uses this component to perform some computation for the output \(s\) based on the inputs \(a\) and \(b\). The guarded actions of the program (as derived by our compiler) are shown in the middle column of Figure 1.

Guarded actions are generated for the data and the control flow of the program. The dataflow guarded actions (DGA) are all assignments that occur in the program and determine the values of the declared variables. The control-flow guarded actions (CGA) are actions of the form \((\gamma \Rightarrow \text{next}(\ell) = \tau)\) where \(\gamma\) is a condition that is responsible for moving the control flow at the next point of time to program label \(\ell\).

### 2.2 Satisfiability Modulo Theories

The dependency analysis and corresponding optimization techniques described in this paper are based on Satisfiability Modulo Theories (SMT). The concept of SMT is to check the satisfiability of logical formulas over one or more theories. While a SAT solver computes an answer to the question “Is there an assignment to the variables of a propositional formula such that the formula evaluates to true?”, an SMT solver is more powerful. It can answer for a formula over first-order predicates of a particular decidable theory (or a combination of such theories) whether there exists an assignment to the variables such that the whole formula evaluates to true.

Such decidable theories (commonly called background theories in the context of SMT) are for example integer linear arithmetic, real linear arithmetic, theories of bit-vectors or arrays [22].

As a result, an SMT solver is capable of more powerful logics than a SAT solver. For example, deciding the satisfiability of the formula \((x > 3) \land (x < 0)\) where \(x\) is an integer, a SAT solver has to abstract each of the inequations by a Boolean variable and derives the formula \((a \land b)\). This formula is obviously satisfiable, as assigning \(a\) and \(b\) both to true makes the whole formula evaluate to true. However, for an SMT solver, it can capture the semantics of the relations > and < and interpret 3, 0 and \(x\) in the domain of integers. A decision procedure for the corresponding background theory will then deliver the appropriate answer that both clauses of the formula above cannot be fulfilled at the same time. Hence the SMT solver returns the result invalid, which means that there is no assignment to the free variables of a formula that makes it satisfiable.

Other answers an SMT solver usually gives are valid, which means that the formula is always true, whatever values are chosen for the free variables, while satisfiable indicates that the truth of the formula depends on the assignment of the free variables. If the background theory is undecidable and if the (thereby incomplete) decision procedure of the background theory fails, it can also return unknown.

### 3. Extended Finite State Machines

This section describes extended finite state machines (EFSMs) that are generated for a set of synchronous guarded actions. Our optimizations will work on EFSMs as internal representation of synchronous programs. After some introductory explanations and definitions given in Section 3.1, Section 3.2 shows how we generate EFSMs from synchronous guarded actions.
3.1 Definition

In order to generate fast sequential code from synchronous programs, the extended finite-state machine (EFSM) representation of the program is an ideal starting point. EFSMs explicitly represent the state space of the control-flow: each state $s$ represents a subset of labels $\mathcal{L}$ of the control-flow labels, and edges between states are labeled with conditions that must be fulfilled to reach the target state from the source state. EFSMs are therefore a representation where the control-flow of a program state is explicitly represented, while the dataflow is still represented symbolically (while synchronous guarded actions represent control flow and dataflow symbolically).

The guards of the guarded actions of the control flow are therefore translated to transition conditions of the EFSM’s transition relations. The guarded actions of the dataflow are first copied to each state of the EFSM, and are then partially evaluated according to the values of the control-flow labels in that EFSM state. Hence, in each macro step, the generated code will only consider a subset $D(s)$ of the guarded actions, which generally speeds up the execution (since many of them are only active in a small number of states).

**Definition 1.** (Extended Finite State Machine). An Extended Finite State Machine (EFSM) is a tuple $(S, s_0, T, D)$, where $S$ is a set of states, $s_0 \in S$ is the initial state, and $T \subseteq (S + C + S)$ is a finite transition relation where $C$ is the set of transition conditions. $D$ is a mapping $S \rightarrow \mathcal{D}$, which assigns each state $s \in S$ a set of datalflow guarded actions $D(s) \subseteq \mathcal{D}$ which are executed in state $s$.

An EFSM is usually illustrated by a directed graph, whose vertices are the states and whose edges are the transitions labeled with the transition conditions. For our example, this graph is given in the rightmost column of Figure 1. In addition to the initial state, three states (combinations of labels) are reachable: $\{w_2, w_3\}$, $\{w_3, w_4\}$ and $\{w_4, w_2\}$.

As its control-flow state is explicit, this model may suffer from state-space explosion since $n$ control-flow locations may result to $2^n$ EFSM states (all locations can be reached simultaneously). It is not only the amount of (control-flow) states that poses problems, but the guarded actions for the dataflow must be also replicated. However, for many practical examples, the EFSM size is still manageable (as a rule of thumb, the EFSM representation is usually about 10 to 100 times larger than the original representation of guarded actions), while providing the fastest target code at the end.

It is important to see the difference between an EFSM and control-flow graphs used in classic compiler design, which also has implications to the optimizations presented in the following section. While ‘states’ of classic control-dataflow graphs consist of assignments that are sequentially executed, states of the EFSM contain still guarded actions that are concurrently executed within one macro step. Moreover, transitions in the EFSM terminate a macro step of the synchronous model, so that new values of the input variables are read on the transition. Due to these differences, many transformations made in classic code optimization cannot directly be applied on EFSMs for code generation of synchronous programs.

3.2 Generation from Guarded Actions

According to the definition of the previous section, an EFSM state specifies which program locations are active in a step. Thus, a state can be seen as a particular assignment to the location variables $\mathcal{L}$. If a location variable $\ell \in \mathcal{L}$ is assigned true, it means in this EFSM state, the control flow of the program is at $\ell$.

A naive way to generate the EFSM states is to take all possible $2^{|\mathcal{L}|}$ states, and compute the transitions from each one. However, as many states are generally not reachable, the algorithm always needs exponential time, even for programs that lead to compact EFSMs. Therefore, a better way to compute the EFSM form is an abstract simulation of the program [16] which basically works as follows.

We explore the reachable control-flow states in a depth-first traversal. For the initial state, we assume that all location variables are set to false. To compute the successor states, this assignment is used for the evaluation of the guards of all CGAs.
function CaseSplitTrue(\(F, PF\))
if \(F = \emptyset\)
return \((\emptyset, \emptyset, \emptyset)\)
forall \(p\phi \in PF\)
\(p\phi \leftarrow \bot\)
choose \(\phi \in F\)
\(p\phi \leftarrow \text{true};\)
\(F := F - \{\phi\};\)
forall \(\phi' \in F\)
if \([\phi \land \phi'] = \text{false}\)
\(p\phi' \leftarrow \text{false};\)
\(P_F := P_F - p\phi';\)
\(F' := F - \phi';\)
\(P_F' := P_F' \cup p\phi';\)
return \((P_F', P_F', F)\)

Figure 2. EFSM generation: conventional case splitting

\((\gamma \Rightarrow \text{next}(\ell) = \text{true})\), which determine the location variables in the next reaction step. If a guard \(\gamma\) is evaluated to true, we know that the corresponding location variable \(\ell\) is assigned true at the next reaction step. Otherwise, if the guard is evaluated to false, the location variable \(\ell\) will not be active at the next reaction step. After all successor states have been generated, we apply the same procedure to new successor states. This traversal of the control-flow states finally creates all reachable EFSM states. Termination is guaranteed because the number of location variables is finite.

However, a simple evaluation of a CGA \((\gamma \Rightarrow \text{next}(\ell) = \text{true})\) might fail: First, \(\gamma\) might contain input variables, which are not known at compile time. Second, it may contain non-Boolean operators and variables, which are still symbolically represented by the EFSM. In these cases, we have to take all possibilities into consideration to make a pessimistic estimation of the reachable states. Assume that \(m \leq n\) is the number of labels whose values cannot be determined. In this case, there are possibly \(2^m\) successor states, which is a very pessimistic approximation. By considering the relationships between the guards of all CGAs, we usually derive a more precise set of successor states. To this end, we make a case split on all remaining variables in the guards \(\gamma\) of the CGAs. Since we do not want to make a case split over non-Boolean variables, we abstract from all non-Boolean predicates appearing as subterms in \(\gamma\) (in the spirit of abstract interpretation [10]) by abbreviating them by fresh Boolean variables. A stepwise case split allows us to finally evaluate the guarded actions for all labels \(\ell \in \mathcal{L}\). The pseudo code in Figure 2 illustrates our approach, where the function takes two parameters: \(F\) the set of renamed Boolean formulas whose truth values are not fixed, and \(P_F\) the corresponding Boolean variables indicating the truth value of the location variables. After one iteration of the case splitting, the fixed Boolean assignment of \(P_F' \subseteq P_F\), the still unfixed set of variables \(P_F\) and the corresponding Boolean formulas \(F\) are returned for the further case splits.

After the generation of the states and transition relations, the DGAs can be distributed to the states. This time, the assignment of the location variables in each state is used to evaluate the truth value of the guards of all DGAs. If the guard is evaluated to false, we can be sure that the DGA is never executed in the state and omit it. Otherwise, we add the DGA to \(D(s)\). This concludes the conventional approach to generate an EFSM.

Figure 5 (c) illustrates the process of propositional case splitting of a set of CGAs given in (a) in the form of a decision tree. First, the set of CGAs (a) are renamed to (b), so that all guards of CGAs are pure Boolean formulas consisting of newly imported Boolean variables. Then, the case splitting begins with variable \(p\). When the truth value of \(p\) is fixed, the truth value of \(lp\) is fixed at the same time. The truth values of \(w1\) and \(w2\) can be derived together in one case distinction. However, the truth value of \(p\) provides no help for deciding the truth value of \(q\). It is still necessary to consider both cases of \(q\) to determine the truth value of \(q3\).

In Figure 1, the rightmost column shows the EFSM generated by using propositional case splitting.

4. Local EFSM Optimizations

4.1 SMT-Based State Optimization

The previous section already explained how EFSM states are basically created. Each time a new control-flow state \(s\) is discovered, the DGAs are instantiated for this particular state (and stored as \(D(s)\)). While the conventional approach can filter dead code very roughly, we can examine the actions of state in more detail with the help of an SMT solver. Our goal is to identify more actions, which can be statically evaluated (constants) or safely removed (dead code). Hence, we are interested in reducing the number of DGAs \(D(s)\) and simplifying all remaining ones.

Due to the synchronous model of computation, the DGAs within an EFSM state are synchronously executed. Therefore, in order to analyze the data dependencies, our optimization can encode them into a single equation system. As each guarded action \(A = (\gamma \Rightarrow x = \tau)\) can be seen as a conditional equation, it can be basically encoded in an implication of the form \(I(A) = \gamma \Rightarrow x = \tau\). Then, it is straightforward to collect the conjunction of all the DGAs within the EFSM state. We call this conjunction the assertion system of the state: \(T_s = \bigwedge_{A \in D(s)} I(A)\).

As described in Section 2.1, if all DGAs assigning \(x\) within a state \(s\) are all eliminated, then the value of \(x\) is determined by the reaction to absence. If \(x\) is declared as event variable, it is assigned a default value. In this case, the reaction to absence can be also added to the assertion system. Hence, this logical description reflects all constraints within a state, and it can be handed over to an SMT solver for checking its consistency.

We check the satisfiability of the guard of each guarded action \(A \in D(s)\) by the SMT solver. If the SMT solver proves that its guard \(\gamma\) is valid, the corresponding action \(A\) is executed every time the state is reached. In this case, we can replace the original guard \(\gamma\) simply by true - in the assertion system as well as in the EFSM so that the redundant check is not executed at run-time. If the SMT solver determines that \(\gamma\) is invalid, we can safely remove \(A\) from \(D(s)\) since it is provably dead code.

The decisions of the SMT solver are based on its knowledge about its background theories. Thereby, our optimization is not only able to deduce many arithmetic correlations between actions, but we also get valuable information about array indices, which generally offers a rich potential for optimization (for details see [2]).

The optimization of the states is directly integrated in the EFSM construction explained in the previous section. Each time, a new state \(s\) is discovered, we encode the assignment of the labels directly into the assertion system of the state \(s\) so that the the DGAs are automatically filtered for the given state \(s\).

In Figure 3, (a) illustrates an EFSM generated by propositional case splitting. By using SMT-based state optimization, DGAs in state \(\{w2, w1\}\) can be simplified. In particular, \(op = \text{false}\) is
propagated inside the state, which leads to the elimination of the DGA \((\text{top} \Rightarrow a = t_2)\).

4.2 SMT-Based Edge Optimization

The previous section considered the DGA's \(D(s)\) of a given state \(s\), which represent the dataflow of the program. In this section, we consider the control-flow, which is originally given by the control flow guarded actions (CGAs) and encoded in the outgoing edges of state \(s\). Similar to the approach presented above, our optimization tries to eliminate as many edges as possible. Thereby, the run-time of the program is improved since redundant checks are eliminated and, much more important, eliminating false edges in the course of the construction may reduce the size of the EFSM, which is a crucial point in the code generation process. Hence, the SMT solver is used to prove that certain EFSM states are unreachable, which justifies their removal. Our methods might therefore also be very useful for the detection of false paths in worst-case execution time analysis [13, 14].

The conventional EFSM generation abstracts non-Boolean predicates by variables. By utilizing an SMT solver, we can examine the truth value of a particular formula at the level of predicates so that we do not need to abstract from them. Let \(\Gamma^c\) be the set of CGAs under examination as described in the Section 3.2. This time, we take the assertion system of the dataflow as described above and directly check the validity of the guard \(\gamma\) for each CGA \((\gamma \Rightarrow \text{next}(t) = \tau)\).

As already experienced in the conventional approach, the SMT solver might not decide for each guards whether it is valid or invalid. For some of them, the given information is insufficient so that a case split is unavoidable for the SMT-based method, too. As the algorithm in Figure 4 illustrates, we take an arbitrary predicate \(\phi\) which cannot be evaluated in the current situation make a case split split on it. This means that we create two derived assertion systems that additionally contain either the clause \(\phi\) or the clause \(\lnot \phi\).

Obviously, if the algorithm of the previous section determined that \(t^*\) will be active (or inactive) in the following step, the SMT solver will determine that \(t^*\) is valid (or invalid). However, by using an SMT solver it is possible to precisely check the satisfiability of formulas, as long as the predicates in the formula are of the decidable theories supported by the solver. The SMT-based case distinction covers more cases than the propositional case distinction described before, as it takes first-order predicates into consideration.

For example, as Figure 5 illustrates, assuming \(x > 5\) the SMT solver can deduce that \(x > 3\) is valid, which eliminates false transitions created by the approach of the previous section, which abstracts from these predicates.

Figure 3 (c) shows the SMT-based edge optimization performed on (b). In particular, the transition condition \((t_1 < t_2)\) is proved by an SMT solver to be valid, while the negation to be invalid.

\[
\begin{align*}
\text{function } \text{CaseSplitTrueSMT}(\Gamma, \mathcal{P}) & \\
\text{if } \Gamma = \emptyset & \\
\text{return } (\emptyset, \emptyset, \emptyset) & \\
\text{forall } p_0 \in \mathcal{P} & \\
p_0 \leftarrow \perp & \\
\text{choose } \phi \in \Gamma & \\
\text{Assert}(\phi); & \\
\Gamma^c := \Gamma - \phi; & \\
\mathcal{P}_t := \mathcal{P}_t - p_0; & \\
\mathcal{P}_{t'} := \{p_0\}; & \\
\text{forall } \phi' \in \Gamma^c & \\
\text{if } [\phi \rightarrow \phi'] = \text{valid} \text{ or } & \\
\mathcal{P}_{t'} := \mathcal{P}_{t'} - p_{\phi'}; & \\
\text{else if } [\phi \rightarrow \phi'] = \text{invalid} & \\
p_{\phi'} \leftarrow \perp; & \\
\mathcal{P}_{t'} := \mathcal{P}_{t'} - p_{\phi'}; & \\
\Gamma' := \Gamma - \phi'; & \\
\mathcal{P}_{t'} := \mathcal{P}_{t'} \cup p_{\phi'}; & \\
\text{return } (\mathcal{P}_t, \mathcal{P}_{t'}, \Gamma) & \\
\end{align*}
\]
(x > 5) => next(w1) = true 
!(x > 5) => next(w2) = true 
(x > 3) => next(w3) = true 
p => next(w1) = true 
!p => next(w2) = true

Figure 5. EFSM states generation: (a) a set of CGAs, (b) propositional abstraction, (c) propositional case splitting, (d) SMT-based splitting

This helps identifying the unreachable state \{w4, w1\}, which is removed in (c).

5. Global EFSM Optimizations

In this section, we consider the global optimization of EFSMs, i.e. optimizations which need to have the full structure for their optimizations. Therefore, they cannot be integrated in the EFSM generation but they are run on the EFSM obtained by the procedures presented in the previous section.

The basic principle of the following algorithms is to analyze how information is propagated across EFSM states. As discussed in Section 2.1, there are only two possibilities how the value of a variable in a step depends on some values of predecessor states. First, memorized variables keep their values unless an action overwrites them. Thus, if no action writes the variable in the current step, it maintains the value of the previous step. Second, delayed actions such as \(\gamma \Rightarrow \text{next}(x) = \tau\) evaluate the right-hand side \(\tau\) in the current step, and write \(x\) only in the following step. Thus, its value depends on the values the variable occurring in \(\tau\) have in the current step.

5.1 Conventional Constant Propagation

Constant propagation is a classic technique which compilers use to optimize a program. It is a static analysis which tries to identify expressions that are constant in all runs of the program. As there is generally not enough information to decide at compile-time whether particular DGA is executed or not, we have to conservatively approximate the propagated constants. The term propagation results from the fact that, once all subexpressions of an expression are shown to be constant, the whole expression is also constant. In our setting, constants have one of the following three origins:

- **type-1**: constants occurring in program expressions, e.g. the literal 5 in \(x < 5\)
- **type-2**: variables assigned by DGAs whose guard is always true in the current state and where the right-hand side evaluates to a constant \(c\), e.g. \(x \in \gamma \Rightarrow x = \tau\) if \(\gamma = true\) and \(\tau = c\)
- **type-3**: event variables set by the reaction to absence, i.e. if the guards of all DGA assigning to \(x\) are always false in the current state, then event variables are reset to their constant default value

To propagate constants within an EFSM state, a so-called environment, i.e. a map assigning constants to program variables, is used. Initially, this map is empty, and each time a variable is evaluated to a constant, the environment is updated correspondingly. By a local fixpoint computation over the evaluation of the DGAs, more and more constants can be computed, and the related DGAs can be simplified accordingly.

In addition to the propagation within an EFSM state (which we call intra-state constant propagation in the following), we can also propagate constants across states (inter-state constant propagation). Hence, the optimizer maintains an environment for each EFSM step. As mentioned above, environments of successor states are linked by delayed actions and the reaction to absence. Hence, if a delayed action \(\gamma \Rightarrow \text{next}(x) = \tau\) is guaranteed to be executed in a given EFSM state \(\gamma\) is always true) and if its right-hand side \(\tau\) evaluates to a constant \(c\), we can propagate this to the successor state and set \(x\) in its environment to \(c\). If a state has several predecessors, they have to agree on the constant \(c\). This is also the reason why this optimization cannot be run in the course of the EFSM generation, where the set of predecessors for a given state might not be complete.

This constant propagation can be embedded in a global fixpoint iteration, which resembles the classic framework for global program optimization by Kildall [21]. With the additional constants given by the predecessor states, there might be more optimizations within a state. When a fixpoint is reached, the DGAs of all states can be optimized accordingly.

Figure 8 (b) shows the result performing constant propagation on (a). In addition to the SMT-based state optimization, the fact \(ap = true\) is propagated to state \(\{w3, w1\}\), which leads to further optimizations in the state.

5.2 SMT-Based Constraint Propagation

In conventional constant propagation, it is difficult to propagate anything more than a constant due to the lack of a adequate mechanism to analyze the semantics of program expressions. For example, consider that in one EFSM state we have the DGA \(true \Rightarrow x = 2*i\). While it is clear that \(x\) will have an even value, an analysis based on simple constant propagation cannot use that information.
Similarly, if delayed actions of several predecessors disagree on a constants for a variable \( x \), no information is derived.

Similar to the previous section, some deficiencies are eliminated in our SMT-based approach. Again, we directly encode the DGAs of an EFSM state \( s \) as clauses of an assertion system \( T_s \), which encodes as much as possible information. Hence, no matter whether the evaluation of a DGA results to a constant value or not, we add the implication \( \gamma \rightarrow x = \tau \) for the DGA \((\gamma \Rightarrow x = \tau)\). Thereby, our approach naturally extends conventional constant propagation as described above, since constants are encoded by the simple constraint \( x = c \).

For inter-state propagation, we need to combine the assertion systems of several EFSM states, since the clauses of predecessor states must be integrated in the assertion system of the current state. A naive way to add the additional constraints from the predecessor state to the successor state is to simply add the clauses to the assertion system of the successor state. However, this does not work as the additional clauses from the predecessor generally contain variables that already occur in the current state. While sharing the same name, the value used for the evaluation of the predecessor constraints should be the value of the variable in the previous state. However, the assertion system of the current state depends on the value of the variables in the current state. Therefore it is necessary to distinguish the variables appearing in the additional constraints and the variables in the original assertion system.

In principle, two strategies can be used to solve this problem. (1) We can use existential quantification over the free variable of the additional constraints to hide them in the combined assertion system. Thereby, we can use the structure of the right-hand side expressions of the DGAs without mess up variables names. Thus, there are no shared variables between the additional constraints and the original assertion system, and both parts can be safely merged. (2) We add a fixed index to all variables in each state, so that variables have a unique name. This strategy does not work for states with outgoing transitions to themselves, i.e. self-loops. For such states, it is necessary to unroll the loop once so that the self-loops are eliminated. However, this doubles the EFSM in the worst case.

Additional problems arise if the constraints should be propagated over several steps, and not only a single then. Strategy (1) cannot be extended to this general case, while strategy (2) forces us to unroll loops up to the size of the desired propagation depth. Therefore, our optimization procedure uses a blend of both strategies, which combines the advantages of both. We first partition the EFSM into extended basic blocks [25]. An extended basic block (EBB) is a sub-tree of the EFSM, where the root of the EBB can either be the initial state of the EFSM, or a so-called join state, which has more than one predecessor. Consequently, the leaves of the EBB are states either do not have successors or join states as successors. This partitioning of the EFSM gives us two important properties, which avoid the problems related to the presented strategies: within an EBB, there are no loops and each internal state has a unique predecessor.

Therefore, we can build an assertion system for an EBB in the style of strategy (2) as follows:

1. **Add indices**: we choose an arbitrary injective indexing function \( I \), which assigns a unique index to each EFSM state. This index is used to rename the DGAs of the EFSM state by substituting each variable \( x \) by \( x^i \).
2. **Handle delayed actions**: Since a delayed action commits the assignment in the following step, we replace \( \text{next}(x^i) \) by \( x^j \), where \( j \) is the index of the (unique) successor of state \( i \).
3. **Combine assertion systems**: For each state, we first build a separate assertion system \( T \). Then, we disjunct the assertion systems of branches. These systems are merged by conjunction with the assertion system of the root to a single \( \Gamma \) for the whole EBB.

To link the assertion systems of the EBBs we follow strategy (1) and first combine all predecessor EBBs by \( \bigvee \Gamma_i \), where each \( \Gamma_i \) is the assertion system of the EBB, which the predecessors of the root state belongs to. This is obviously a conservative approximation, since we enter each EBB by only one predecessor.

For example, in Figure 1 in the generated EFSM, the **initial state** and the state \( \{w2, w1\} \) form an EBB, while state \( \{w3, w1\} \) and state \( \{w4, w1\} \) form two EBBs respectively. Anyway, if the two transitions from \( \{w3, w1\} \) and \( \{w4, w1\} \) to \( \{w2, w1\} \) are eliminated, the whole EFSM forms a single EBB. Figure 6 shows the corresponding assertion system under this assumption.

By checking the satisfiability of all guards in the context of the assertion systems of the EBBs, the EFSM can be further simplified. As for conventional constant propagation, we embed the whole procedure in a fixpoint iteration, which repeatedly simplifies the EFSM and checks the satisfiability of actions and evaluation of constants. The complete algorithm doing the fixpoint computation is sketched in Figure 7.

Figure 8 shows an example for our constraints propagation. The optimization (c) is based on the ‘propositionally’ generated EFSM (a). It is seen that all constant propagation done in (b) are also performed in (c). Furthermore, as the delayed assignments of \( i1 \) and \( i2 \) are encoded in the EBB’s assertion system, it is clear that

\[
\begin{align*}
\Gamma_1 &= ( \\
& (\text{true} \Rightarrow t1 = i1 + i2 + 3) \land \\
& (\text{true} \Rightarrow t2 = i1 + i2 + 5) \land \\
& (\text{true} \Rightarrow o1 = i1) \land \\
& (\text{true} \Rightarrow i1 = o1) \land \\
& (\text{true} \Rightarrow i2 = o2) \land \\
& (\text{true} \Rightarrow \text{op}_1 = \text{false}) \land \\
& ) \\
\Gamma_2 &= ( \\
& (\text{true} \Rightarrow i1 = o1) \land \\
& (\text{true} \Rightarrow i2 = o1) \land \\
& (\text{true} \Rightarrow \text{op}_2 = \text{true}) \land \\
& (\text{true} \Rightarrow t1 = i1 + i2) \land \\
& (\text{true} \Rightarrow t2 = i1 + i2) \land \\
& (\text{true} \Rightarrow o2 = t1) \land \\
& (\text{true} \Rightarrow o2 = t2) \land \\
& (\text{true} \Rightarrow s2 = s1) \land \\
& ) \\
\Gamma_3 &= ( \\
& (\text{true} \Rightarrow i1 = o1) \land \\
& (\text{true} \Rightarrow i2 = s1) \land \\
& (\text{true} \Rightarrow \text{op}_2 = \text{true}) \land \\
& (\text{true} \Rightarrow t1 = i1 + i2) \land \\
& (\text{true} \Rightarrow t2 = i1 + i2) \land \\
& (\text{true} \Rightarrow o2 = t1) \land \\
& (\text{true} \Rightarrow o2 = t2) \land \\
& (\text{true} \Rightarrow s2 = s1 + 1) \land \\
& ) \\
\Gamma_i &= \Gamma_1 \land (\Gamma_2 \lor \Gamma_3 )
\end{align*}
\]
function ConstraintPropagation(S, s0, T, D)
    do
        G := graph of (S, s0, T, D)
        change := false
        forall EBB_i ∈ EBBs of (S, s0, T, D)
            Γ_i := AssertionSystem(EBB_i)
            for all l ∈ TransitionRelations(EBB_i)
                p := Γ_i ∧ condition_i
                if Check(p) = valid
                    condition_i := true
                else if Check(p) = invalid
                    condition_i := false
                G’ := Refine(S, s0, T, D)
                if G’ ≠ G
                    change := true
        while change

Figure 7. Algorithm of constraint propagation in EBBs

i1 < i2 can be identified as invalid. By reaction to absence, s is
assigned 0.

6. Experimental Results

We evaluated all the optimizations presented in the previous
sections with the help of some practical examples. Our F# imple-
mentation is based on our Averest system\(^2\), and the SMT solver Z3 [24],
which was directly connected via its .NET managed API.

In total, we took 28 examples from the Averest benchmark suite.
All examples are given as synchronous guarded actions as XML-
based AIF (Averest Intermediate Format) files, which have been
compiled from synchronous Quartz programs. Our optimization
procedure reads such an AIF file and writes the optimized result
in another AIF file. In between, it applies the SMT-based optimiza-
tions presented in this paper: the local state and edge optimiza-
tions and the global constraint propagation. For all of them, the optimization
procedures returned a result within 5 to 10 minutes on a standard
desktop computer. For the original and the optimized version,
we generate C code, which is subsequently compiled by gcc. For both variants,
we chose the target arm-elf (ARM processor, ELF binary format), a common platform for embedded systems. By enabling
all optimizations (compiler switch -O3), we check whether gcc can also find the optimizations our procedure has found.

Our first metric is the size of the intermediate code and thereby
the size of the two EFSMs: the original EFSM was only generated
with propositional case splitting, and the EFSM with the SMT-
based local and global optimizations. As our experiments show,
there is correlation between size and runtime performance. While
preserving the general structure of the EFSM, reduction of size
leads to a reduction of time, since computations of the program
are eliminated. Please note that the optimizations introduced in this
paper do not increase neither size nor runtime as the procedures
only eliminate computations.

Our second metric is the actual worst-case execution time
(WCET), which we obtained from the WCET analyzer Bound-T\(^3\). Bound-T determines for architecture-specific executables a safe
upper bound for the number of CPU cycles the program requires
on a specified processor. There are several versions of Bound-T,
which support various target platforms; we used the version for
ARM7 processors. For each benchmark, we used Bound-T to
determine the WCET for a single step of the synchronous program
(i.e. the worst-case reaction time (WCRT)).

Table 9 shows the results for our examples. The column headers
should be self-explanatory. Empty cells in the table indicate that we

\(^2\)http://www.averest.org

\(^3\)See http://bound-t.com.
have not carried out the WCET analysis, since there is no difference between the original AIF file and the optimized file.

More than half of the examples (15 of 28) have been improved by applying our optimization procedures. Size reductions vary from some kBs to couples of MBs. The results of the WCET analysis also show that the examples generally have better performance with our optimizations: the WCRT is improved by 5%–10% for most examples.

For Island Traffic Control, the unoptimized version contains a lot of unreachable and inconsistent states, which are discovered by the invariant inference. Light Control System is a very small system with rudimentary dataflow, where our optimizations do not produce improved code. Logical Sensors is the sensor fusion part of a vehicle stability system. It consists of many small components like (CarVelocity), which are responsible for the computation of a dedicated sensor value. The complete sensor fusion LogicalSensors consists of too many EFSM states for a global optimization. Thus, the three variants contain different partitions. Again, one can see the potential of our optimization: While one component like CarVelocity and other components as well (not shown in the table) may have no improvements in size reduction, the different partitions of the LogicalSensors module always have a size reduction. By taking a look at the source code of LogicalSensors, we noticed that the components are working in parallel. During the EFSM generation, there are a lot of false transitions, which are detected and removed by our optimization. The four examples BuildPixel, IDTC, Dequantization and DeZigZag are all subparts of a JPEG decoder. Two of them show positive results for our optimizations. Since a large amount of computations in these modules are matrix calculations, our optimizations can exploit array theory and detect the redundant codes within. The final three examples show the potential of our optimization for small programs. As the name suggests, Warshall implements Warshall’s algorithm for computing the transitive closure of a graph, and SpiralSFG is parallel variant for systolic arrays. Finally FIFO is the realization of a simple FIFO buffer.

One thing noticeable here is that, although the WCRT of some examples have no explicit improvement, the average WCET might still be improved. As the generated C code is basically a large switch-case nested in a while-loop with each case representing an execution of a state in the EFSM, the WCRT of the program is simply the case that takes longest to execute. The reduction of the WCRT in turn depends on the optimization of the “heaviest” state. However, this state might rarely be executed in practice. If we know the frequency of execution of each state, we can then calculate the average WCET of the program.

To summarize, we made the following observations: (1) Even small programs can be optimized further. (2) The larger the programs, the more optimizations can be typically made. This is often the case due to large parallel loops, where the original EFSM contains lots of false transitions. These transitions can be eliminated, which causes a chain effect that leads to further reductions of the system. (3) The larger the dataflow part of the guarded action is, the larger are the SMT-based optimizations.

### 7. Conclusion

In this paper, we presented a set of optimization techniques for synchronous programs that are based on a modern SMT solver. By using an SMT solver, our optimization techniques remain at the abstraction level of the program expressions and are therefore able to determine more optimizations as compared to abstract interpretation to boolean formulas. All optimizations essentially implement a dataflow analysis for extended finite state machines. Our local optimization collects information within a particular state during the EFSM generation and aims at elimination of unreachable states. Our global optimization is performed after the EFSM generation, and utilizes the structural information of the entire EFSM. Our experiments have clearly shown that our optimization techniques are able to further optimize code generated from synchronous programs.

### References


---

### Table: Experimental Results

<table>
<thead>
<tr>
<th>Example</th>
<th>dead code elimination</th>
<th>passive code elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AIF WCRT</td>
<td>AIF WCRT</td>
</tr>
<tr>
<td>Island Traffic Control</td>
<td>4,444 kB 1,466</td>
<td>3,095 kB 1,462</td>
</tr>
<tr>
<td>Light Control System</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Cruise Control</td>
<td>58</td>
<td>48</td>
</tr>
<tr>
<td>Car Velocity</td>
<td>142</td>
<td>142</td>
</tr>
<tr>
<td>Logical Sensors A</td>
<td>2,469 2,628</td>
<td>2,241 2,628</td>
</tr>
<tr>
<td>Logical Sensors B</td>
<td>8,022 7,483</td>
<td>7,839 7,482</td>
</tr>
<tr>
<td>Logical Sensors C</td>
<td>24,758 7,566</td>
<td>23,953 7,478</td>
</tr>
<tr>
<td>BuildPixel</td>
<td>2,852 12,914</td>
<td>2,839 12,903</td>
</tr>
<tr>
<td>IDCT</td>
<td>8,038 45,281</td>
<td>7,614 41,202</td>
</tr>
<tr>
<td>Dequantization</td>
<td>721</td>
<td>721</td>
</tr>
<tr>
<td>DeZigZag</td>
<td>434</td>
<td>434</td>
</tr>
<tr>
<td>Warshall</td>
<td>32</td>
<td>305</td>
</tr>
<tr>
<td>SpiralSFG</td>
<td>129</td>
<td>479</td>
</tr>
<tr>
<td>FIFO</td>
<td>35</td>
<td>180</td>
</tr>
</tbody>
</table>

Figure 9. Experimental Results


