Abstract—In this paper, we present a static data-flow analysis for synchronous programs, which is used to improve the run-time efficiency of the generated code. Our optimization techniques are based on extended finite state machines (EFSMs) which are obtained by a translation from synchronous guarded actions. In order to avoid a potential state space explosion of that translation, the subsequent analysis does not necessarily consider monolithic EFSMs, but is able to cope with a set of concurrent EFSMs, where invariants of the individual parts are collected and shared. Experiments show the potential of the optimizations: optimized programs generally have a smaller size and a better run-time performance.

I. INTRODUCTION

Synchronous languages have proved to be very useful for the modeling and implementation of reactive systems. Compared to alternatives, the underlying synchronous model of computation has many advantages, in particular, a formally defined semantics with a deterministic concurrency, which simplifies the static analysis and the verification of systems. Furthermore, the synchronous model of computation simplifies programming, since developers do not have to care about low-level details like timing, synchronization and scheduling. However, the synchronous paradigm has some consequences that make the compilation of synchronous programs not at all straightforward: In particular, causality analysis [27, 29] and schizophrenia problems [27, 29] challenge compilers.

Research over the last two decades has tackled these problems, and various compilers have been developed [3, 21, 27] based on different code generation schemes like automaton based code, (boolean) equation systems and concurrent control/data-flow graphs. While the first articles on compilation of synchronous programs focused on the correctness of the compilers, mainly considering the semantic issues like schizophrenia and causality problems, new research efforts consider the efficiency of the generated code: more and more compilers were designed with a target architecture in mind, which provides a better basis for optimization.

This paper also aims at optimizing synchronous programs, in particular, the run-time performance of the generated code. The main contribution of this paper is a customized optimization for synchronous systems, which combines many techniques from different areas: we adapt methods from classical compiler design (such as constant propagation) and verification (such as on-the-fly model checking), enrich them by concepts designated for synchronous languages (such as elimination of passive code), and employ state-of-the art SMT-solving (as provided by Z3) as a decision procedure.

The code optimization considered in this article is not based on a particular source language. Instead, we aim at optimizing an intermediate code based on synchronous guarded actions, which are a target-independent intermediate format that can be used for all synchronous languages. Thus, our optimization techniques can be applied to several source languages and is independent of the later on chosen code generation scheme or target architecture.

Data-flow analysis has been a static analysis tool for code optimization in classical compiler design for decades. Early work has been presented in [1, 15, 16, 22] and considers different kinds of data-flow 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 data-flow analysis is a well-established tool for classic code optimization, it has not yet been widely used for the compilers of synchronous languages. The static analysis described in [28] is used to compute conditions for instantaneous execution of a synchronous program that is done similarly by the control flow predicates in [23]. In [6], we developed a static data-flow analysis based on fixpoint computations similar to the classic data-flow analyses, which will be the basis for one stage of the optimization technique presented in this paper.

The rest of this paper is structured as follows: Section II first describes the starting point of our optimization, namely synchronous guarded actions, and sketches extended finite state machines (EFSMs), which are the basis for our optimizations. Section III forms the core of our paper: it presents our optimization strategies, starting from optimizing single EFSM to finally optimizing concurrent EFSMs. Section IV
shows experimental results, before we conclude with a short summary in Section V.

II. SYNCHRONOUS SYSTEMS

A. 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.
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 [5, 7, 24].

Synchronous guarded actions are designed in the spirit
of classical guarded commands [9, 13, 17], 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), which postulates that a system is ex-
cuted 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. 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 \( \mathcal{V} \). 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 \( \gamma \Rightarrow x = \tau \) (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 EventVars(\( \mathcal{V} \)) \( \subseteq \mathcal{V} \) are reset to their default values
(like wires in hardware circuits) by the reaction to absence,
while memorized variables MemVars(\( \mathcal{V} \)) \( \subseteq \mathcal{V} \) store their
previous values (like registers in hardware circuits).

The left-hand side of Figure 1 gives an example, which
consists of two threads. The first one implements a simple
arithmetic component that can compute two different func-
tions \( t_1 \) 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.

B. Extended Finite State Machines

Guarded actions are generated for the data and the con-
rol flow of the program. The data-flow 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 \( \gamma \Rightarrow \text{next}(\ell) = \text{true} \) where \( \gamma \) is a condition that is responsible
for moving the control flow at the next point of time to
program label \( \ell \in \mathcal{L} \).

In order to generate fast sequential code from synchronous
programs, the extended finite-state machine (EFSM) repre-
sentation of the program can be used. It makes the state
space of the control-flow explicit: each state \( s \) represents
a subset Labels(\( s \)) \( \subseteq \mathcal{L} \) of the control-flow labels, and
edges between states are labelled with conditions that must
be fulfilled to reach the target state from the source state.
EFSMs are therefore a representation where the control-flow
part of a program state is explicitly represented, while the
data-flow part is still symbolically represented.

The guards of the CGAs are therefore translated to
transition conditions of the EFSM’s state transitions. The
DGAs are first copied to each state of the EFSM, and 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 \( T \subseteq (S \times C \times S) \) is a finite set of transition
relations where \( C \) is the set of transition conditions. \( D \) is a
mapping \( S \rightarrow D \), which assigns each state \( s \in S \) a set of
DGAs \( D(s) \subseteq D \) which are executed in state \( s \).

An EFSM is usually illustrated by a directed graph, whose
vertices are the states and whose the 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: \( \{w_2, w_1\} \), \( \{w_3, w_1\} \) and \( \{w_4, w_1\} \).
The guarded actions are partially evaluated according to this
module MAC (int{256} \(a, b, s\)) {
  int{256} \(i_1, i_2, o, t_1, t_2\);
  bool \(op\);
  loop {
    \(w_1: \text{pause}\);
    \(t_1 = \ldots\)
  }
}

It is important to see the difference between an EFSM and control-flow graphs of classical compiler design, which also has implications to the optimizations presented in the following section. While ‘states’ of classic control-data-flow graphs consist of assignments that are executed sequentially, 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 classical code optimizations cannot directly be applied on EFSMs for code generation of synchronous programs.

III. OPTIMIZATIONS

This section introduces our optimization techniques that aim at reducing the size of the EFSM – the number of the states and transitions, as well as the number of the DGAs of each state, and thereby, they improve the runtime of the code. The optimizations can be partitioned into two categories: while constant propagation (Section III-A) and invariant inference (Section III-B) both eliminate dead code (unreachable code), the optimization of Section III-C eliminates passive code (redundant code). While all these techniques optimize a single EFSM, some discussion on optimizing concurrently running EFSMs are drawn in Section III-D.

Due to the synchronous MoC, the guarded actions within a macro-step can be seen as a set of conditional equations, and all possible valid executions must satisfy the equations of the visited states. Hence, we can formulate the task of dead code detection as a satisfiability problem. If the guard of an action cannot be satisfied, the action can be safely removed. Thereby, we assume that the given set of guarded actions is causally correct [2, 25, 26] so that the corresponding theories are consistent. To reason about these equation systems, we use the SMT (Satisfiability Modulo Theories) solver Z3 [12]. Formally, this tool tells us whether a formula of a first-order theory is valid, satisfiable, or invalid. Our first-order formulas are given by the program expressions, i.e., bounded naturals, integers, and arrays. Thereby, we can keep the level of abstraction high, while providing deep insights into the semantics of the given guarded actions. Before discussing the details of the optimization techniques, we give some definitions that will be used through out the rest of the paper.

**Definition 2 (Guarded Action Variables):** Let \(FV(\tau)\) denote the free variables occurring in the expression \(\tau\). Then, the following definitions determine the sets of variables which are referenced in different parts of a guarded action. \(\text{grdVars}(\cdot)\) contains all variables of the guard, \(\text{rhsVars}(\cdot)\) all variables occurring on the right-hand of the assignment,
rdVars(·) all variables read, and wrVars(·) all variables written by the guarded action.

\[
\begin{align*}
\text{grdVars} (\gamma \Rightarrow x = \tau) & := \text{FV}(\gamma) \\
\text{grdVars} (\gamma \Rightarrow \text{next}(x) = \tau) & := \text{FV}(\gamma) \\
\text{rhsVars} (\gamma \Rightarrow x = \tau) & := \text{FV}(\tau) \\
\text{rhsVars} (\gamma \Rightarrow \text{next}(x) = \tau) & := \text{FV}(\tau) \\
\text{rdVars} (\gamma \Rightarrow x = \tau) & := \text{FV}(\gamma) \cup \text{FV}(\tau) \\
\text{rdVars} (\gamma \Rightarrow \text{next}(x) = \tau) & := \text{FV}(\gamma) \cup \text{FV}(\tau) \\
\text{wrVars} (\gamma \Rightarrow x = \tau) & := \{x\} \\
\text{wrVars} (\gamma \Rightarrow \text{next}(x) = \tau) & := \{\text{next}(x)\}
\end{align*}
\]

A. Constant Propagation

The first optimization technique that we will consider in this section is constant propagation. The idea resembles classic constant propagation techniques and is relatively straightforward – we collect and propagate the constants within an EFSM state and subsequently use them for satisfiability checking of its outgoing state transitions and DGAs. Guards that are never satisfiable indicate dead code, which can be safely removed.

Basically, constant propagation aims at creating static environments for each EFSM state, which are similar to the lattices used in sequential program flow analysis [18, 20].

Definition 3 (Environment): For a given EFSM over variables \( V = \{x_1, \ldots, x_n\} \), where \( x_i \) has domain \( D_i \), an environment \( \mathcal{E} \) of a state is defined as a mapping of variables to their domains, i.e., for \( x_i \in V, \mathcal{E}(x_i) = v \) with \( v \in D_i \).

For a particular EFSM state, there may be more than one environment that conforms to the state and its DGAs, since the inputs are not known at compile time, and many local and output variables depend on the input values. Formally, a congruence relation can be defined on the environments of an EFSM state as the environments modulo the constants in that state.

The goal of constant propagation is to gather useful information of a given state, and to complete the corresponding environment as much as possible. In the context of synchronous languages, we can derive information from three different sources within a state. For example, consider an EFSM state that contains the following DGAs:

\[
\begin{align*}
\text{true} & \Rightarrow x = 1 \\
x < 5 & \Rightarrow y = 10 \\
x > 3 & \Rightarrow z = \text{true}
\end{align*}
\]

For them, we can deduce the following constants:

1) type-1: constants that are directly encoded in expressions: e.g., in the guards \((x < 5)\) and \((x > 3)\), 5 and 3 are directly encoded constants, which can be immediately used.

2) type-2: constants that can be deduced from actions: e.g., \(x = 1\) will be always executed. This could help us deduce that condition \((x < 5)\) is valid, which in turn makes \(y = 10\) a valid equation.

3) type-3: constants that can be deduced from the reaction of absence: e.g., since \(x = 1\), we know that \((x > 3)\) is never satisfiable, which means that \(z = \text{true}\) will never be executed. As there is no action setting \(z\), we know that the reaction to absence always determines the variable. If \(z\) is an event variable, it will be always false: if \(z\) is a memorized variable, it will be the previous value.

Note that there is a difference between the computation of the type-2 and type-3 constants. There are many environments for an EFSM state, and constants should be the same in every environment of the same congruence class. Thus, we need to check for all environments whether there is some action whose guard is valid for type-2 constants, while all actions must have invalid guards for type-3 constants.

Our constant propagation algorithm basically follows Kildall’s scheme for classical framework for global program optimization [18]. The difference is that for each EFSM state, an additional inner fixpoint is computed for its constants. These constants are then used for the propagation, which is the outer fixpoint computation.

Figure 2 gives the pseudo code. Constants are collected in the form of equations. If the guard of action \(A : \text{var} = c\) is valid and \(c\) a constant, the equation \(\text{var} = c\) is then added to the set of constants. The function CollectConst collects the constants of the form \(\text{next}(x) = c\) and transforms them to the form as \(x = c\). The constants agreed on the predecessors of \(s\) are then joined to the constants of \(s\). The function
Check calls an SMT solver to check the satisfiability of the given formula based on the context of the constants \( p \). The procedure ReactToAbs checks whether all guarded actions having an assignment to the variable written by \( A \) have invalid guards. If so, it assigns to the variable a value according to its type. For an event type, the value assigned is a constant. For a memorized type, it depends on the constants of the predecessors. If the state’s predecessors can agree on a constant value, then it is assigned a constant as well.

Finally, transitions with invalid transition conditions are eliminated. If all the transitions to a state are thereby removed, the state is unreachable, and is therefore removed.

For example, consider our previous example of Figure 1. We apply the procedure described above to the unoptimized EFSM shown in Figure 3 (a). The propagation of the constant values of \( op \) from state \( (w_2, w_1) \) to its two successors is used to simplify some guarded actions (e.g. \( \langle op \Rightarrow o = t_1 \rangle \) in state \( (w_3, w_1) \) becomes \( \langle true \Rightarrow o = t_1 \rangle \)) and to remove some others (e.g. \( \langle \neg op \Rightarrow o = t_2 \rangle \) in state \( (w_3, w_1) \)).

### B. Invariant Inference

In this section, a more powerful method is presented, namely invariant inference. In general, compared with constant propagation, invariant inference finds more optimizations, but requires more computation. In practice, it is up to the programmer to choose which technique to use.

Basically, the method collects all constraints given by the guarded actions of the EFSM to build an assertion system. For each state, we can define its assertion system as follows:

**Definition 4 (Assertion System of an EFSM state):**

Let \( \langle \gamma \Rightarrow A \rangle \in D(s) \) a guarded action of a given EFSM state \( s \). Its corresponding implication \( I(\gamma \Rightarrow A) \) is \( \gamma \Rightarrow x = \tau \) if \( A \) is the immediate action \( x = \tau \), or \( \gamma \Rightarrow \text{next}(x) = \tau \) if \( A \) is the delayed action \( \text{next}(x) = \tau \).

The assertion system \( \Gamma_s \) of \( s \) is then

\[
\Gamma_s := \bigwedge_{\langle \gamma \Rightarrow A \rangle \in D(s)} I(\gamma \Rightarrow A)
\]

Then guards can be directly checked against the assertion system. Invariant inference goes one step further, in which it does not look at each EFSM state separately, but it looks at several states at the same time in order to get more information from the state transitions. Thus, we compose the assertion systems of several states to get an integrated view of the examined states. Since variables may have different values in each state, we need to rename them so that they can be distinguished in a composed assertion system.

However, it is generally impracticable to build a single assertion system for the whole EFSM. Due to cyclic transitions, states may be visited more than once in a run. Hence, renaming variables by states is not enough, and loops would have to be unrolled (infinitely often if we want to discover all facts). Since finding appropriate bounds for unrolling is practically impossible, we follow a different approach and restrict the satisfiability problem to non-cyclic sub-structures of the original EFSM, i.e. the extended basic blocks [20].

**Definition 5 (Extended Basic Block):** An extended basic block of an EFSM \( (s_0, S, T, d) \) is a structure that contains a subset of the states \( S_{ebb} \subseteq S \) and transitions between them \( T_{ebb} \subseteq (S_{ebb} \times B \times S_{ebb}) \cap T \) such that they form a tree. The root is either the initial state \( s_0 \) or a vertex with an in-degree originally greater than 1, while all the other nodes have in-degree 1. The leaves either have an original out-degree 0, or their successors have in-degree greater than 1.

For example, Figure 4 shows a an EFSM and its EBBs.

![Figure 4. An EFSM graph and its extended basic blocks](image)

Then, the assertion system for an EBBs can be defined as follows.

**Definition 6 ((Combined) Assertion System of an EBB):** Let \( B \) an arbitrary extended basic block with the set of states \( S_b \). Let \( \Gamma_s \) be the assertion system for state \( s \in S_b \). The assertion system of \( B \) is built by the following steps:

- For the renaming, we use an arbitrary injective indexing function \( i \) with signature \( S_b \rightarrow \mathbb{N} \), which assigns a unique index to each state. Then, let \( \phi_i(s) = [x_1^{i(s)}, \ldots, x_j^{i(s)}] \) be the renamed guarded actions, where all occurrences of a variable \( x \) have been substituted by \( x^i \).

- For a renamed delayed guarded action \( \phi_i(s) \), in order to prevent name conflicts, remove it from \( \Gamma_s \) and add it to the assertion systems of all successor states of \( s \).

- The assertion system of an EBB \( \Gamma_B \) is the combined assertion system \( \Gamma_{r_B} \) of the root state \( r \). Provided that the successors of \( s \) are \( s' \in S' \) with transition condition \( \tau_{s 
arrow \tau_{s' \to s'} \Rightarrow [x_1^{i(s')}, \ldots, x_j^{i(s')}] \) be the renamed transition condition. The combined assertion system of a state \( s \) is recursively defined as follows:
do
G := graph of \((S, s_0, T, D)\)
change := false
forall \(EBB_i \in EBBs\) of \((S, s_0, T, D)\)
\(\Gamma_i := \text{AssertionSystem}(EBB_i)\)
forall \(t \in \text{TransitionRelations}(EBB_i)\)
\(p := \Gamma_i \land \text{condition}_t\)
if Check(p) = valid
\(\text{condition}_{t'} := true\)
else if Check(p) = invalid
\(\text{condition}_{t'} := false\)
\(G' := \text{Refine}(S, s_0, T, D)\)
if \(G' \neq G\)
change := true
while(change = true)

Figure 5. Invariant inference

\[ \Gamma' := \bigvee_{s' \in S'} \left( \Gamma_i \land \tau_i(s) \right) \]

where each \(\Gamma_i(s)\) is the assertion system of state \(s\) generated by the first two steps and \(\Gamma'\) the combined assertion system of state \(s\).

The resulting combined assertion system is then used to check the satisfiability of guards and transition conditions in the EBB. For the roots of EBBs, we take the disjunction of the assertion systems of its predecessors. This is a pessimistic, but conservative approximation, since some of the predecessors might be unreachable.

The pseudo code in Figure 5 illustrates our complete invariant inference algorithm. It iteratively tests the satisfiability of the transition conditions in EFSM based on the assertion systems of the EBBs. Function AssertionSystem builds the assertion system of the EBB according to Definition 6. Function TransitionRelations collects all transition relations within an EBB. After each iteration, function Refine removes all unreachable states. This process continues until there are no further unreachable states. The satisfiability of DGAs in each state can then be checked, without changing the shape of the EFSM.

Figure III shows the result of invariant inference for our example. In addition to the constants already found in the previous approach, the relation \(r1 < r2\) is identified, and state \((w_4, w_1)\) is recognized as unreachable and therefore removed.

C. Passive Code

The synchronous model of computation requires that in every step, all inputs are read, and all outputs are synchronously computed as the reaction of the program. Additionally, internal variables are synchronously updated. However, not all values of the internal variables are required in every reaction step to compute the outputs. For this reason, our optimization procedure computes for every internal variable a condition that determines whether its value is required for the current or future computation of outputs. In this sense, our optimizations allow us to identify so called passive code, which can be disabled to avoid unnecessary computations.

In [6], we presented a method to symbolically determine passive code of synchronous programs. To this end, we defined the predicate \(\text{req}_i\) for every output and local variable \(x\), which should hold when the value of \(x\) is required for the computation of the current macro step.
Obviously, the required conditions define a don’t care set of states: whenever \( \neg \text{req}_x \) holds, an action setting \( x \) can be either executed or not, the observable result will be the same. As already pointed out in [5, 8], just adding \( \text{req}_x \) to the guards of all actions does not lead to an improvement. It generally worsens the efficiency of the program since dynamically evaluating the required condition may cost more than simply executing the action. In practice, this tradeoff between the evaluation of the guard and the action is very important for the optimization.

This section adopts the given approach and uses a precomputed predicates to optimize the given EFSM. Thus, we first compute the required predicates for all variables as described in [5]. Then, we use these predicates in the assertion systems for code optimization. In these systems, we strengthen the guard of an action \( (\gamma \Rightarrow x = \tau) \) by \( \text{req}_x \), i.e. we use \( (\gamma \wedge \text{req}_x) \Rightarrow x = \tau \) in the analysis. Similar to the approach presented in Section III-B, if the analysis finds out that the guard is invalid and that it is never executed, we remove the action from the EFSM. However, if the guard is still satisfiable, we keep the original action \( (\gamma \Rightarrow x = \tau) \). Hence, we use \( \text{req}_x \) only statically so that no additional overhead is introduced to the program and the optimized code is guaranteed to have a better run-time performance.

Figure 3 (d) shows the results of the analysis for our running example. It is seen that in state \( (w_3, w_1) \) the computation of the output \( o \) is set according to \( t_1 \). Thus, \( t_2 \) contains a redundant value, and its last assignment is eliminated. The changes in state \( (w_3, w_1) \) trigger further optimizations in state \( (w_2, w_1) \).

D. On-the-fly Optimization

A single monolithic EFSM generally produces the fastest code for a given set of synchronous guarded actions. However, as its control-flow state is explicit, this model suffers from state-space explosion since \( n \) control-flow labels may result to \( 2^n \) EFSM states. It is not only the amount of (control-flow) states that poses problems, but the guarded actions for the data flow must be also replicated. (As a rule of thumb, the EFSM representation is usually about 10 to 100 times larger than the original guarded actions). Therefore, we now consider a set of concurrently running EFSMs instead of single monolithic EFSM (in the degenerated case, there is an EFSM for each control-flow label, which leads to a symbolic code generation similar to [2] for Esterel).

Our general workflow is as follows: we partition a given system according to a given structure (e.g. its original module structure in the source code), create an EFSM for each and optimize them locally according to the methods presented in the previous sections. Then, in a second pass, we optimize every EFSMs again, now by using information from the concurrently running parts.

Due to the synchronous model of computation, concurrent EFSMs run in lockstep, i.e. their states and transitions are synchronized. The product of the EFSMs is based on a single assignment of variables so that the values of the shared variables must be consistent in each step. Since we already have assertion systems for each local EFSM state, the computation of synchronized states, i.e. which local EFSM states are active at the same time, is the key to sharing information between EFSMs and to derive more knowledge for optimization of the program.

Figure 6 shows a simple example illustrating how shared information can be exploited for optimization. The synchronized states of EFSMs (a) and (b) are simply \( \{s_1, s_3\} \) and \( \{s_2, s_4\} \). By local optimizations, there is no way to determine whether the guard \( (X > 5) \) in state \( s_4 \) is valid, which implies the validity of \( Y = true \). Ideally, if we know that \( s_2 \) and \( s_4 \) are synchronized, we can easily add the information of \( X = 10 \) to the assertion system of the other state, and the fact \( Y = true \) can be derived straightforwardly. In general, if we know the synchronized states of any other concurrent EFSM, we can additionally use the disjunction of their assertion systems for optimization.

Hence, to share the computed results between EFSM requires that we know which set of states in the EFSMs are synchronized. However, computing the synchronized states is equivalent to computing the synchronized product of the EFSM [10]. However, we have the opinion (similar to [11]) that the bottleneck of global optimization is rather the lack of memory and not the lack of time. While our algorithm basically walks through the whole product, it does not create this model in memory. Hence, while the optimization might take some time, it is still manageable.

As the name suggests, this approach is inspired by the on-the-fly model checking techniques [4], which do not precompute the whole transition system before the actual checking, but compute the structure when needed. Since any invalid transition relation leads to unreachability of its target state, in the coupled EFSM, any elimination of the transition relations within a coupled EFSM benefits from the computation of the synchronous product as well.

With the information of synchronized states, we build an assertion system for each local EFSM to check the
satisfiability of the transition relations in the other local EFSMs. As the exact set of synchronized states depends on inputs, we can only compute an approximation statically:

Let \( Q = \{ q_1, \ldots , q_k \} \) be the states such that whenever \( s \) is executed, \( q_1, \ldots , q_k \) might also be executed, and \( Q_1, \ldots , Q_k \) the corresponding assertion systems. Let \( Q' = \{ q'_1, \ldots , q'_k \} \) be a superset of \( Q \), and \( Q'_1, \ldots , Q'_k \) be the corresponding assertion systems. Then, we can still derive:

\[
\text{Check}(\bigvee V_i Q'_i \land \phi) = \text{valid} \\
\Rightarrow \text{Check}(\bigvee V_i Q_i \land \phi) = \text{valid} \\
\text{Check}(\bigvee V_i Q'_i \land \phi) = \text{invalid} \\
\Rightarrow \text{Check}(\bigvee V_i Q_i \land \phi) = \text{invalid}
\]

The algorithm computing such a superset of synchronized states is given in Figure 7. Function BuildAssertSys takes the disjunction of all the assertion systems of the synchronized states of \( p_i \) from the concurrently running EFSMs. Function Successors computes the successive synchronized states of state \( \{ p_1, \ldots , p_n \} \) according to the transition relations of each EFSM \( (S_i, s^0_i, T_i, D_i) \).

IV. EXPERIMENTAL RESULTS

We evaluated all the optimizations presented in the previous section on some practical examples. Our F# implementation is based on the Averest system\(^1\), and the SMT solver Z3 [19]

\(^1\)www.averest.org

that was directly connected via its .NET managed API. All tested systems were given as synchronous guarded actions in AIF (Averest Intermediate Format) files, which have been compiled from Quartz programs. Our optimization reads such an AIF file and writes the optimized result in another AIF file. Thereby, we used two optimizations: one based on invariant inference (IVIF) and another one using invariant inference and passive code identification (IVIF+IDPC) together. Similar to our small example in Figure 3, we record the size of the AIF files - and thereby of the following four EFSMs: the original EFSM without any optimization (corresponds to (a)), with invariant inference (corresponds to (c)), and additionally identification of passive code (corresponds to (d)). Finally, the right-most column shows the proportions of the reductions from the original AIF files to the fully optimized counterparts. Each of the given examples has been optimized within 5 minutes.

As already mentioned, the evaluation is based on the size of the optimized AIF files: the smaller the size, the better the optimization. The reason to choose size rather than runtime as the metric of comparison is that the run-time of the programs varies if inputs vary. Since it is difficult to get appropriate data for testing, the reduction of the size is a good generic criterion. 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.

Table 8 shows the results of applying our optimization procedures to various examples ranged from some kBs to dozens of MBs. For Island Traffic Control, the unoptimized version contains a lot unreachable and inconsistent states, which are discovered by the invariant inference. Light Control System is a very small system, where our optimizations do not produce improved code. Finally, the last example is a part of a vehicle stability system, namely its sensor fusion component Logical Sensors.

The programs WheelsFree, WheelsSlip, WheelsVelocity, XAcceleration and YawRate are all parts of this component, which are responsible for the computation of a dedicated sensor value. The complete component LogicalSensors consists of too many states for a global optimization. Thus, the idea here is to split it into smaller programs. The three versions v1, v2 and v3 are different configurations that only contain different partitions of the sub-modules. Again, one can see the potential of our optimization.

In general, we can conclude the following observations:

(1) Dead code and passive code elimination together usually achieve better results. (2) The larger the examples are, the more potential for optimization they have. This is partly due to large parallel loops, where the original EFSM contains lots of false transitions. These transitions can be eliminated,
Programs | Original [KB] | IVIF [KB] | IVIF+IDPC [KB] | Reduction
--- | --- | --- | --- | ---
Island Traffic Control | 60,863 | 3,407 | 3,407 | 94.4%
Light Control System | 25 | 25 | 25 | 0%
Wheels Free | 18 | 18 | 18 | 0%
Wheels Slip | 75 | 75 | 75 | 0%
Wheels Velocity | 338 | 338 | 338 | 0%
XAcceleration | 27 | 21 | 21 | 22.2%
YawRate | 384 | 119 | 119 | 69.0%
Logical Sensors v1 | 33,712 | 33,188 | 30,915 | 8.2%
Logical Sensors v2 | 4,817 | 4,639 | 4,163 | 13.5%
Logical Sensors v3 | 493 | 430 | 263 | 46.6%

Figure 8. Experimental results

which causes a chain effect that leads to a dramatic reduction of the system.

Finally, it is interesting to compare our customized optimization with the general data-flow analysis of gcc. For this, we generated C code from the original and the optimized AIF files, and let GCC compile and optimize (switch -O3) both variants. By comparing the generated assembler code, we check whether GCC found the same optimizations.

However, as gcc’s optimization procedures change the structure of the code and thereby increase its size, a comparison of the code size as done above does not make sense. Instead, we illustrate the additional power of our optimizations with the help of a small example. Figure 9 shows four C code fragments generated by our compiler for different synchronous programs. In all code fragments, $a$ and $b$ are input variables, $s$ is an output and $t_1$, $t_2$ are local variables of the original synchronous program. In the C code, they are all locally declared in the scope of a function so that gcc is able to optimize computations on them.

Fragment (a) shows an example before optimization: two delayed assignments of local variables are executed in state 1. In state 2, these two variables are compared, and the output is assigned to one of them based on the comparison. As shown in (b), invariant inference recognizes that the comparison in state 2 is invalid, and the whole if is replaced by a simple assignment. gcc does not see this optimization. Fragment (c) gives another example, where two instantaneous assignments are executed for the local variables in state 1. In (d), passive code identification removes an assignment that is not required, while gcc basically retains the computations of (c).

Hence, we cannot simply rely on the data-flow analysis of gcc to optimize code derived from synchronous programs. In addition to these differences, our optimization also benefits from the additional power of SMT, and it does not get disturbed by artifacts of the code generation (more C variables than program variables).

V. CONCLUSION

In this paper, a set of optimization techniques for synchronous programs was developed. By using an SMT solver, the optimization technique keeps the abstraction level to predicates over program expressions. All optimizations basically implement a data-flow analysis for Extended Finite State Machines. Constant propagation and invariant inference are used to find dead code, and passive code elimination targets local computations that are irrelevant for the outputs. Experiments have shown the potential of the optimization techniques. There is still some future work left, in particular related to the optimization of concurrently running EFSMs. While better approximations for the synchronized states would give more information for the optimization, a general shift to hierarchical EFSMs instead of parallel one might also be another direction to explore.

REFERENCES

int a, b, t1, t2, s;
...
unsigned int state = 1;
while(1) {
    switch (state) {
        case 1: {
            printf("%d", s);
        } break;
        case 2: {
            printf("%d", s);
        } break;
    }
}
Figure 9. Comparison with gcc: (a) without IVIF, (b) with IVIF, (c) without IVIF+IDPC (d) with IVIF+IDPC