Compositional Equivalence Checking for Models and Code of Control Systems

Rupak Majumdar  Indranil Saha  Koichi Ueda  Hakan Yazarel

Abstract—We present CSEC (Compositional Symbolic Equivalence Checker), an algorithm and tool to perform automatic and compositional equivalence checking of C code against Simulink models. Such equivalence checking is important in model-based development of safety-critical control software in industrial settings, where either the Simulink models are hand-generated to correspond to existing legacy code bases, or the C code is generated from Simulink models using code generators. In the former case, the manual translations may not preserve behavior; in the latter case, equivalence checking is necessary to ensure that the code generator has not introduced bugs if the code generator itself is not certified. CSEC compositionally constructs proofs of equivalence of two call graphs, by constructing a formula that is valid if two functions are equivalent, when all called functions are assumed equivalent. The validity of the formula is checked using an SMT solver. We have applied CSEC to a module of powertrain controller C code base and the corresponding semi-automatically translated Simulink model at Toyota, and have automatically uncovered several dissimilar behaviors between models and code. We have also applied CSEC to prove equivalence of a Clutch Lockup Model and the automatically generated C code from the model.

I. INTRODUCTION

There is a significant trend in embedded safety-critical systems design toward model-based development (MBD). In MBD, higher-level models of the system are developed and simulated, and execution code is automatically generated from the models. The shift to a higher-level modeling language allows the domain expert to focus on the problem without concerns for lower-level implementation details, thereby improving productivity. The code generator provides the glue between the models and the implementation concerns relating to the execution platform. Several software development guidelines, such as RTCA DO-178C [8] in the avionics domain, voice their support for MBD.

The shift to MBD is not a panacea to all productivity and reliability issues. One particular issue is equivalence verification between models and code, in order to ensure that the functionality captured in the models is maintained by the executable code. While code generators hold the promise for correct-by-construction translations from models to code, in practice, they may not be provably correct and can introduce bugs in the translation process. In addition, during the shift to MBD, legacy code subsystems co-exist with newly developed models, and some of the models are obtained by hand-translating legacy code to the modeling language. In fact, safety standards such as DO-178C [8] and ISO 26262 [12] recommend verification of equivalence between models and code.

In this paper, we describe Compositional Symbolic Equivalence Checker, CSEC for short, a tool to automatically perform equivalence verification between Simulink blocks and C code, as well as our experience in applying the tool to verify the equivalence between hand-generated models and legacy code at Toyota. We focus on Simulink 1 [23], a high-level modeling tool from MathWorks, as it is the de facto standard for model-based development of embedded control systems. For example, Toyota uses Simulink to build high-level models for their next generation systems.

CSEC is based on compositional analysis of the code and the model, and symbolic execution and SMT solving for each component. We exploit the close correspondence between the model and code. Our algorithm proceeds bottom-up on the call graphs of the model and the code, proving each corresponding pair of functions to be equivalent. It automatically constructs a logical formula from the bodies of the two functions such that the two functions are equivalent iff the formula is valid. Validity is checked using an SMT-solver (Yices [9], in our implementation). In proving two functions \( f \) and \( g \) equivalent, CSEC assumes that each called function in the bodies of \( f \) and \( g \) are equivalent and uninterpreted. This approximation is sound and cuts down on SMT solving time, but can be conservative. However, in our experiments, we found very few false negatives with this approximation.

We applied CSEC to a Simulink model with over 800 Simulink blocks and over 600 lines of C code which implements a module of Toyota’s powertrain controller. The model was generated semi-automatically to match existing legacy C code for experimental purpose [27]. Using CSEC, we found several discrepancies between the model and the code fully automatically and in a few milliseconds for each function in the C program. We also applied CSEC to prove equivalence of a Simulink model available in Simulink demos provided by MathWorks, and the C code generated automatically from the model by Real-Time Workshop (RTW, currently named as Simulink Coder) [24] code generator.

While the foundations of compositional equivalence

1Simulink is a registered trademark of The MathWorks, Inc.
void moduleMainFunction(void)
{
    int local_in;
    local_in = global_var >> 4;
    if (local_in > 255)
    {
        local_in = 255;
    }
    moduleSubFunction(local_in);
}

Our objective is to show that the control program in Figure 1 and the moduleMainFunction subsystem in the Simulink model in Figure 2 are equivalent. We adopt a compositional strategy. We first generate function call graphs for both the C program and the Simulink model and check if they are isomorphic. In our example, the Simulink model and the C code have isomorphic function call graphs. The C program has four functions: moduleMainFunction, moduleSubFunction, moduleSubFunction1 and moduleSubFunction2. The corresponding Simulink subsystems also have the same names.

Our algorithm checks equivalence for each pair of corresponding functions in the C code and the Simulink model. When proving equivalence for a pair of functions, we consider any function call in the body of these functions to be uninterpreted, that is, we only assume that equal inputs to these functions produce equal outputs, but nothing further about the properties of these outputs.

To prove equivalence of a function in the C code and the corresponding subsystem in the Simulink model, we ask the following question: if the corresponding inputs of the C function and the Simulink subsystem take the same value, is
it possible that their output would be different? We cast this problem as a satisfiability problem for a set of constraints involving four parts: (a) the corresponding inputs to the C function and the Simulink subsystem are equal, (b) the strongest postcondition relating the inputs and the outputs of the C code, (c) the strongest postcondition relating the inputs and the outputs of the Simulink subsystem, and (d) a constraint stating that the outputs of the C function and the Simulink subsystem are different. We provide details on the computation of strongest post condition in the next section. If this set of constraints is unsatisfiable, we conclude that the function in the C code and the corresponding subsystem in the Simulink model are equivalent.

As an example, to prove the equivalence of the function moduleSubFunction1 and the corresponding subsystem, we solve the set of constraints shown in Figure 3. An “c” postfix is added to each variable in the C code and an “s” postfix is added to each variable in the Simulink model. If the constraints set is unsatisfiable, then it can never be the case that given the corresponding inputs of the C function and the Simulink subsystem are equal, the outputs differ from each other. And hence we conclude that the function and the subsystem are equivalent.

To prove equivalence of a function that calls other functions we treat the called functions to be uninterpreted functions. For example, the function moduleSubFunction calls moduleSubFunction1 and moduleSubFunction2. To prove the equivalence of the function moduleSubFunction in the C code and the subsystem moduleSubFunction in the Simulink model, we assume that the function moduleSubFunction1 (moduleSubFunction2) in the code is equivalent to the subsystem moduleSubFunction1 (moduleSubFunction2) in the model, and thus treat them as one uninterpreted function. Figure 4 shows the set of constraints to prove the equivalence of the function moduleSubFunction and the subsystem moduleSubFunction. Note that there is no return variable in the function moduleSubFunction in the C code, however we have to make sure that the global variable global_var takes the equal value at the exit point in the C function and in the corresponding Simulink Subsystem. If there are multiple global variables updated in a function or one return variable with one or more global variables, we replace the last constraint by the disjunction of individual “output distinct” constraints for each of such variables.

In the set of constraints in Figure 4, moduleSubFunction1 and moduleSubFunction2 are two uninterpreted functions. Both of them take one integer input and return one integer output. In the set of constraints, we have presented a constraint generated by a function $f$ called with arguments $a_1 \ldots a_n$ as $f(a_1 \ldots a_n)$.

If a called function does not return a value, it generally updates some global variables. For example, the void function moduleSubFunction updates the global variable global_var. When we call such a function from another function, the output of the uninterpreted function corresponding to the called function corresponds to the global variable. If there are multiple global variables updated in a function or one return variable with one or more global variables, we need to define multiple uninterpreted functions one for each such variable. The function moduleMainFunction calls the void function moduleSubFunction and the set of constraints in Figure 5 is used to prove the equivalence of the function moduleMainFunction and the corresponding Simulink subsystem moduleMainFunction.

As in a C function one variable may be assigned for more than once, we convert a C function to its static single assignment form [16] before we generate constraints from it. For example, in moduleMainFunction the variable local_main has been assigned for two times. For the second assignment we add a suffix “2” to the end of the variable name in the corresponding constraint.
As we have already proved that the two function call graphs proving equivalence of two functions, we treat all function congruent functions in the two function call graphs. While elligently conclude that the two programs are not equiv-

Given two programs, we verify their equivalence com-

III. EQUIVALENCE VERIFICATION

Given two programs, we verify their equivalence com-

TABLE I

are isomorphic, the congruent functions in the two programs call the same set of functions.

A. Representation of Program Functions

We define simple arithmetic programs over real-valued variables. A simple arithmetic function \( F = (I, G, L, O, s) \) consists of a set \( I \) of input variables, a set \( G \) of global variables, a set \( L \) of local variables, a set \( O \) of output variables, and a body \( s \) defined by the following grammar:

\[
s := x := c \mid x := y \oplus z \mid x := \text{call } f(y_1, \ldots, y_n) \mid (1)
\]

where \( x \) ranges over variables in \( G \cup L \cup O \), \( y, z \) range over variables in \( I \cup G \cup L \), \( c \) ranges over arbitrary rational constants, \( e \) ranges over Boolean expressions over the variables in \( I \cup G \cup L \), and \( \oplus \in \{+,-,\ast\} \) ranges over arithmetic operations. The body \( s \) consists of assignment statements, function calls, sequential composition \( s_1 ; s_2 \), conditionals \( \text{assume}(e) \), and non-deterministic choice \( s_1 \mid s_2 \).

The conditions and non-deterministic choices are used to capture control flow (e.g. if-then-else) in the code. For \( x, z \) variables and \( c \) a rational constant, we write \( x := c \ast z \) as shorthand for \( y := c; x := y \ast z \). We assume \( I, G, L, \) and \( O \) are disjoint sets.

Note that we do not consider loops in the core language. This is because most programs in this domain come with static bounds on the number of iterations of a loop, and so loops can be unrolled statically. We omit the treatment of arrays and structures here; our implementation handles these features.

For a set \( X \) of variables, an \( X \)-valuation is a mapping from \( X \) to real values. We write \( \{(X)\} \) for the set of all \( X \)-valuations. For an \( X \)-valuation \( \nu \) and variable \( x \in X \), we write \( \nu(x) \) for the value of \( x \) under the valuation \( \nu \). A program state is a \( I \cup G \cup L \cup O \)-valuation. In the following, we use first-order formulas with free variables in \( I \cup G \cup L \cup O \) to denote sets of program states: a formula \( \theta \) represents the set of program states that satisfy the formula \( \theta \).

The semantics of a function is given using the strongest post-condition operation \( SP \) [28]. The function \( SP \) maps a body \( s \) and a first-order formula \( \theta \) to a first-order formula \( SP(s, \theta) \). We use the notation \( \theta(x' / x) \) to denote the formula \( \theta \) in which each occurrence of \( x \) is substituted by \( x' \). For a first-order formula \( \theta \) with free variables in \( X \), and an \( X \)-valuation \( \nu \), we say \( \nu \) satisfies \( \theta \) if the formula obtained by substituting each occurrence of \( x \in X \) with \( \nu(x) \) evaluates to true. In the following, we identify an \( X \)-valuation \( \nu \) with a \(|X|\)-dimensional vector. For a first-order formula \( \theta \) and a set.
$X = \{x_1, \ldots, x_k\}$ of variables, we write $\exists X. \theta$ as shorthand for $\exists x_1, \exists x_2, \ldots, \exists x_k. \theta$.

The transformer $SP$ is shown in Table I. The operations have the usual semantics [28], and $SP(s, \theta)$ returns a formula that characterizes the set of program states that can be reached by executing the statement $s$ from a program state satisfying the formula $\theta$.

The semantics of a function defines a mathematical relation, written $[P]$, between $I \cup G$-valuations and $O \cup G$-valuations in the following way: the pair of valuations $(\mu, \nu) \in [P]$ is in the relation if $\mu$ is an $I \cup G$-valuation, $\nu$ is an $O \cup G$-valuation, and $\nu$ satisfies $\exists L. SP(s, \bigwedge_{i \in I \cup G} \nu = \mu(v))$.

### B. Proving Equivalence of Two Functions

Let $F_1 = (I_1, G_1, L_1, O_1, s_1)$ and $F_2 = (I_2, G_2, L_2, O_2, s_2)$ be two functions. We say $F_1$ and $F_2$ are compatible for equivalence if there are bijections $f_i : I_1 \rightarrow I_2$, $f_o : O_1 \rightarrow O_2$, and $f_g : G_1 \rightarrow G_2$ mapping input, output, and global variables of $F_1$ to corresponding variables in $F_2$. Let $F_1$ and $F_2$ be compatible for equivalence, and let $\mu_1$ be an $(I_1 \cup O_1 \cup G_1)$-valuation, for $i \in \{1, 2\}$. We write $\mu_1 \equiv \mu_2$ iff for each $x \in I_1$, we have $\mu_1(x) = \mu_2(f_i(x))$, for each $x \in O_1$, we have $\mu_1(x) = \mu_2(f_o(x))$, and for each $x \in G_1$, we have $\mu_1(x) = \mu_2(f_g(x))$.

We now formally define the equivalence between two functions from two different programs.

**Definition 1:** The *Equivalence decision problem* takes as input two functions $F_1 = (I_1, G_1, L_1, O_1, s_1)$ and $F_2 = (I_2, G_2, L_2, O_2, s_2)$, and bijection $f_i : I_1 \rightarrow I_2$, $f_o : O_1 \rightarrow O_2$, and $f_g : G_1 \rightarrow G_2$, and returns “yes” iff for every pair $\mu_1 \in \{I_1 \cup G_1\}$ and $\mu_2 \in \{I_2 \cup G_2\}$ such that $\mu_1 \equiv \mu_2$, and for every pair $\nu_1 \in \{O_1 \cup G_1\}$ and $\nu_2 \in \{O_2 \cup G_2\}$ satisfying $(\mu_1, \nu_1) \in [F_1]$ and $(\mu_2, \nu_2) \in [F_2]$, we have that $\nu_1 \equiv \nu_2$.

Intuitively, for “equivalent” input valuations $\mu_1$ for function $F_1$ and $\mu_2$ for function $F_2$, we run $F_1$ on $\mu_1$ and $F_2$ on $\mu_2$ and compare the state of the output and global variables. The answer to the equivalence decision problem is “yes” if running the programs with the same inputs (i.e., $\mu_1 \equiv \mu_2$) results in the same outputs (i.e., $\nu_1 \equiv \nu_2$ for the outputs).

The equivalence verification problem reduces to a validity problem in the logic of (non-linear) arithmetic and uninterpreted functions as follows. Consider functions $F_1$ and $F_2$ compatible for equivalence. We construct the following formula:

\[
\bigwedge_{u_1 \in I_1, u_2 \in I_2} u_1 = u_2 \land \bigwedge_{g_1 \in G_1, g_2 \in G_2} g_1 = g_2 \land \forall \nu \in O_1 \cup O_2 : f_o(\nu) = \nu \land \forall \nu' \in L_1 \cup L_2 : f_g(\nu') = \nu'.
\]

In the formula, part (a) encodes the equality constraints on the corresponding variables in $I_1 \cup G_1$ and $I_2 \cup G_2$, (b) encodes the semantics of the functions $F_1$ and $F_2$ respectively, and (d) encodes the equality of the corresponding variables in $O_1 \cup G_1$ and $O_2 \cup G_2$. The answer to the equivalence decision problem is “yes” iff the above formula is valid. We denote that two functions $F_1$ and $F_2$ are equivalent by $F_1 \equiv F_2$.

Note that in constructing the strongest postconditions of the bodies of the two functions, we treat all calls to other functions as uninterpreted functions. For an uninterpreted function $f$, $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$, whenever $x_i = y_i$ for all $i \in \{1, \ldots, n\}$.

The following theorem provides the complexity of proving equivalence of two functions.

**Theorem 1:** The complexity of equivalence checking for two simple arithmetic functions is in PSPACE.

**Proof:** The proof of the theorem follows from the fact that the validity problem for quantifier-free expressions over $(\mathbb{R}, +, \cdot)$ is in PSPACE [5], and this complexity remains the same in the presence of Boolean connectives and uninterpreted functions.

### C. Proving Equivalence of Two Programs

A program $P$ is represented by $P = (G, B, G)$, where $G$ is a set of *global variables*, $B$ is a set of functions where each function takes the set $G$ as its global variables, and $G = (L, l_0, E, w)$ is a directed function call graph of $P$, where $L$ denotes the set of locations that correspond to the functions of $P$, $l_0$ is the location corresponding to the initial function where the execution of the program starts, $E \subseteq (L \times L)$ is the set of edges representing function calls, and the function $w : L \rightarrow B$ is a bijection that maps each location in $L$ to a unique function in the set $B$. Thus, $(u, v) \in E$ means that the function $w(u)$ calls the function $w(v)$.

Let $P_1 = (G_1, B_1, G_1)$ and $P_2 = (G_2, B_2, G_2)$ be two programs, where $G_1 = (L_1, l_{01}, E_1, w_1)$ and $G_2 = (L_2, l_{02}, E_2, w_2)$. We can prove the equivalence of two programs by proving the equivalence of the initial functions in the two programs using the algorithm presented in Section III-B. However, note that we prove equivalence of two functions in two programs under the assumption that all the corresponding called functions are equivalent.

**Proposition 1:** The programs $P_1$ and $P_2$ are equivalent (denoted by $P_1 \equiv P_2$) iff $w_1(l_{01}) \equiv w_2(l_{02})$.

To prove the equivalence of two programs $P_1$ and $P_2$, our algorithm CSEC($P_1, P_2$) makes the following two checks.

1. $G_1$ is isomorphic to $G_2$, i.e., there exists a bijection $f : L_1 \rightarrow L_2$, such that $(l, l') \in E_1$ iff $(f(l), f(l')) \in E_2$.
2. $w_1(l) \equiv w_2(f(l))$ and $w_1(l') \equiv w_2(f(l'))$.

The algorithm returns “yes” if both checks pass, and “don’t know” otherwise. The following theorem ensures the soundness of our algorithm.

**Theorem 2:** If CSEC($P_1, P_2$) returns “yes” then $P_1$ and $P_2$ are equivalent.

The converse of the theorem is not true, because two programs may be equivalent but the approximation made by assuming all called functions are uninterpreted may be too coarse to prove two functions to be equivalent.
IV. THE CSEC TOOL

In this section, we present CSEC that implements the algorithm described in section III. The tool has four main components. The first and the second components parse the C program and the Simulink model respectively and gather necessary information about the variables and functions and generate constraints from the code and the model. The third component uses the information about the functions in the C program and the subsystems in the Simulink model to generate function call graphs for both the program and the model, and check if they are isomorphic. If the function call graphs are isomorphic, the fourth component attempts to prove the equivalence of the congruent functions and subsystems. Below we present each component of the tool.

A. Parsing C Program

We use CIL (C Intermediate Language) [18] to parse a C program. Using CIL we can generate function call information from the C program to generate a function call graph, and the control flow structure of each C function. We use the control flow structure to extract relevant information from a function. By using this information we generate Yices [9] constraints that represent the strongest postcondition for the output of the function.

B. Parsing Simulink Model

To parse a Simulink model, we use Simulink model construction APIs [26] provided by Mathworks, for example, find_system and get_param. Using these APIs, we construct a transition system for the Simulink model. From this transition system we gather relevant information to construct the function call graph, and for each subsystem we generate constraints representing the strongest postcondition in Yices input language.

C. Checking Function Call Graphs Isomorphism

To check if the function call graphs of the C program and the Simulink model are isomorphic, we use the following method. Our assumption is that the names of a function and the corresponding subsystem are the same. We traverse the two function call graphs in lock-step in breath-first manner. When we visit a function in the function call graph of the C program, we check if a subsystem with the same name is present in the function call graph of the Simulink model. While visiting the called functions from a function, we find out if the set of called functions are the same in both the C program and the Simulink model. If at any step, we cannot match the functions, we stop and conclude that the two graphs are not isomorphic, and also the program and the model are not equivalent.

D. Checking Equivalence of a Function-Subsystem Pair

To prove equivalence of a function in the C code and the corresponding subsystem in the Simulink model, we ask the following question: if the corresponding inputs of the C function and the Simulink subsystem take the same value, is it possible that their output would be different? We cast this problem as a satisfiability problem for a set of constraints shown in (2), only the constraint (2d) is modified to capture that the outputs of the C function and the Simulink subsystem are different. We employ the Yices SMT solver to check if the set of constraints is unsatisfiable. If the constraints are unsatisfiable for all function-subsystem pairs, the C program and the Simulink model under consideration are equivalent. If for any instance the constraints are satisfiable, we generate a test case from the generated model by Yices to show the discrepancy.

V. CASE STUDIES

In this section we present two case studies: proving equivalence of (1) a specific module of Toyota powertrain controller legacy C-code and its semi-automatically translated Simulink model, and (2) the Clutch Lockup model available in Simulink demo and RTW generated C code from the model.

A. Toyota’s Powertrain Controller

In this section, we present an industrial case study on a particular Toyota vehicle’s powertrain control system. Toyota has undertaken a project to translate the legacy C code to Simulink models. The Simulink model is developed from the legacy C-code semi-automatically for experimental purpose [27]. The objective of this project is to examine the feasibility of translating Toyota’s legacy code base to Simulink models. One of the biggest challenges in this effort is to verify the equivalence of the legacy C code and translated Simulink model.

The Simulink model conforms to MAAB modeling style guidelines [15]. There are more than 800 Simulink blocks in the model. In the model, there are Simulink standard blocks from Sources, Sinks, Math Operations, Logic and Bit Operations, Ports and Subsystems, Signal Routing, Discrete, Discontinuous and User-Defined Functions libraries. The C code has 24 functions, and there are 7 levels of function call hierarchies.

Our objective is to prove that the Simulink model is behaviorally equivalent to the Legacy C code. As the first step, we prove that the function call graph of the code and the model are equivalent. Next, we attempt to prove a function and its corresponding subsystem to be equivalent. However, if this effort fails, we generate a test case that shows the discrepancy in the behavior of the two. Based on the test case, we fix the discrepancy in the corresponding subsystem in the model. We then again attempt to prove the equivalence between the C function and the model subsystem. We iterate this process until we are able to prove their equivalence.

We have found a number of discrepancies in the Simulink models in this process, before we finally obtain a model that is proven to be equivalent to the legacy C code. To prove the equivalence of a C function and the corresponding Simulink subsystem, CSEC takes less than 20ms in each case.

The discrepancies we have found in the Powertrain Control Simulink model are described below:
• **Type Mismatch** The C code-base uses signed and unsigned variant of short integer, integer and long integer. In Simulink, the corresponding datatypes are “int8”, “uint8”, “int16”, “uint16”, “int32” and “uint32”. In Simulink, the type of a block is set as a signal attribute parameter in the Function Block Parameters pane. We found a number of cases where the datatyp of a Simulink block was not selected correctly.

• **Constant Value Mismatch** The C code contains a number of left shift and right shift operations. In the Simulink model, a left shift operation is implemented as a multiplication operation, and a right shift operation as a division operation. If in the C code we have \( y = x \ll n \) (\( y = x \gg n \)), in the Simulink model we have blocks implementing \( y = x \times 2^n \) (\( y = x/2^n \)). Using CSEC, we have found one scenario, where an instance of \( y = x \gg 4 \) in the C code was implemented as \( y = x/10000 \) instead of \( y = x/16 \). From the test case generated by CSEC, we could easily find out the source of error.

• **Missing Model Component** In the C code, after any arithmetic operation, the result is checked against possible overflow and underflow using if conditions. In case of an overflow (underflow), the output of the operation is set to an upper (lower) bound. Using CSEC, we found a case in the model, where the check for a possible overflow after a multiplication operation was not implemented in the Simulink model.

• **Off-by-one Error** In the Simulink model, for loops are implemented as For Iterator Subsystem. A For Iterator Subsystem contains a For Iterator block inside it, and the initial and final values of the iteration variable are set in the Source Block Parameters pane of the For Iterator block. The initial value of the iteration variable can be set using the “Index Mode” parameter. There are two options for “Index Mode” - “One based” and “Zero based”. In the C code base the iteration variables in the for loops start from 0. However, CSEC found one scenario where in the Simulink model, the “Index Mode” for the For Iterator block was selected as “one based”.

Out of the 24 functions, for two functions CSEC generated false alarm - it generated counterexamples that implied that the C function and the corresponding subsystem are not equivalent. Careful analysis of the counterexamples revealed that the functions and corresponding subsystems were actually equivalent. In one case, the false alarm is due to the fact that in the C function a variable was declared as “short int”. In the Simulink subsystem the same variable is used as a boolean variable. According to the values the variable can actually take, the data type of the variable can be boolean. However, the legacy C code was written in C90 standard which did not support boolean data type in C. The test case generated by CSEC shows that the value chosen for the variable in the C function is greater than 1, but in actual execution of the program the variable can take value either 0 or 1. We manually changed the datatype for the C variable in the yices constraint to “bool”, and then we were able to show the equivalence of the function and the corresponding Simulink subsystem. In the second case, a function that implements variable value saturation has two formal parameters lower_bound and upper_bound that are used as the lower and upper limit of the variable respectively. To prove equivalence of this function and the corresponding subsystem in the Simulink model, CSEC generates a test case that assigns a value to the lower bound that is bigger than the value assigned for the upper bound. The false alarms could be alleviated by adding an invariant lower_bound < upper_bound in the set of constraints.

**B. Clutch Lockup Model**

In this case study, we prove the equivalence of the Clutch Lockup model using if-action subsystem from Simulink demo [25] and its RTW generated C code. The Simulink model captures the dynamics of the clutch system containing two plates that transmit torque between the engine and the transmission. Using Real-Time Workshop, we generate C code with independent functions for all subsystems in the model. The model has 10 subsystems in 3 levels of hierarchy. The model has total 102 blocks, and the generated C code after cleaning the parts unnecessary for equivalence checking is around 250 lines long. We have proved the equivalence of all 10 subsystems and the corresponding C functions. The maximum time required to prove equivalence of a subsystem and its corresponding C function is 7ms.

In the C code, we injected different kinds of errors related to arithmetic and relational operators and constant mismatch in different functions. CSEC catches errors in all the cases and generates a test case in each case in less than 10ms.

**VI. RELATED WORK.**

Equivalence checking is an important component of the verification flow for hardware circuits [13], [14]. In software compilers, equivalence checking is usually called translation validation [17], [2], [22], and is used to check that an optimization pass by a compiler has not introduced any bugs. The seminal paper of Pnueli et al. [20] describes the translation validation of a compiler from a synchronous programming language to C code. Recently, Godlin and Strichmann [11] give proof rules for reasoning about program equivalence that also replace called functions by uninterpreted functions.

The necessity of applying compositional strategies to verify equivalence of industry scale hardware circuits was identified long ago and cutpoint based compositional equivalence verification technique was proposed [3], [4]. Cutpoint based compositional verification strategy was also applied to assembly level embedded software [10]. Our compositional strategy is similar to cutpoint based compositional verification in the sense that both of them divide the equivalence verification problem to a number of small size verification problems. The major difficulties in cutpoint based compositional strategy is to identify proper cutpoints and that it generates false alarm due to not identifying the relationship
between the variables at the cutpoints. In verifying the equivalence of a C program and a Simulink model, CSEC exploits the compositional structure of both the code and the model - the function boundaries become natural choices for the cutpoints. Moreover, the relationship between the input variables of a function is generally known, and can be used as invariant if required.

The problem of verification of equivalence of a Simulink model and the automatically generated code from the model was addressed in a few recent works. The CGV tool from Mathworks [6], [7] employs dynamic testing to show numerical equivalence between model and the generated code. In this approach, the same input stimuli is applied to both the model and the code and the outputs are compared. Though this method is scalable and provides some confidence on the generated code, it does not guarantee soundness on the equivalence of the model and the code. TVS [21] is a tool to prove equivalence between Simulink model and C programs. TVS generates a single verification condition from the Simulink model and RTW generated C code to show the equivalence of the two. On the other hand, CSEC uses compositional technique, thus can be used for model and code of much larger size. CSEC is also useful to prove partial equivalence of the model and code. In case of reverse engineering when model is being generated from a C code base, CSEC can be used to check which part of the code still needs to be translated, and if there is any bug in the already translated model. Moreover, CSEC generates test cases if there is any discrepancy between a function and the corresponding subsystem. Function level test cases are easier to interpret than a test case generated for the full model.

VII. Conclusion

We have reported an algorithm and a tool to verify equivalence of a Simulink model and a C code. Our algorithm is compositional - it is capable of dealing with large industry-size model and code. We have applied our tool to Toyota’s powertrain control system to demonstrate that our tool can be used to prove equivalence of a C code and the Simulink model generated from the code. We have also applied our tool to an example of Clutch Lockup model from Simulink demos provided by MathWorks to demonstrate that our tool is capable of proving equivalence of a Simulink model and automatically generated C code from the model.

In embedded software domain, the C code may be generated using fixed-point arithmetic, introducing quantization error in the output of the function. In this case, we can still use our framework to prove equivalence of the model and the code, but the notion of the equivalence needs to be changed – A C function implemented using fixed-point arithmetic is equivalent to a subsystem in a model using real variables, if the difference of the output of the C function and the model subsystem is bounded above by a specified small constant. To compute strongest post-condition for a C function using fixed-point arithmetic, we can use the method provided by Anta et al. [1].

References


