Hindawi
Scientific Programming
Volume 2020, Article ID 6791891, 19 pages
https://doi.org/10.1155/2020/6791891

Research Article

Improved Efficiency of Object Code Verification Using Statically Abstracted Object Code

N. Shaukat,[1] S. Shuja ,[1] S. K. Srinivasan,[2] and S. Jabeen[1]

[1]*Department of Electrical and Computer Engineering, COMSATS University Islamabad (CUI), Islamabad 44000, Pakistan*
[2]*Department of Electrical and Computer Engineering, North Dakota State University, 1411 Centennial Boulevard, Fargo, ND 58102, USA*

Correspondence should be addressed to S. Shuja; sanashuja@comsats.edu.pk

Received 18 February 2020; Accepted 13 June 2020; Published 14 July 2020

Academic Editor: Autilia Vitiello

One of the major challenges in the formal verification of embedded system software is the complexity and substantially large size of the implementation. The problem becomes crucial when the embedded system is a complex medical device that is executing convoluted algorithms. In refinement-based verification, both specification and implementation are expressed as transition systems. Each behavior of the implementation transition system is matched to the specification transition system with the help of a refinement map. The refinement map can only project those values from the implementation which are responsible for labeling the current state of the system. When the refinement map is applied at the object code level, numerous instructions map to a single state in the specification transition system called stuttering instructions. We use the concept of Static Stuttering Abstraction (SSA) that filters the common multiple segments of stuttering instructions and replaces each segment with a merger. SSA algorithm reduces the implementation state space in embedded software, subsequently decreasing the efforts involved in manual verification with WEB refinement. The algorithm is formally proven for correctness. SSA is implemented on the pacemaker object code to evaluate the effectiveness of abstracted code in verification process. The results helped to establish the fact that, despite code size reduction, the bugs and errors can still be found. We implemented the SSA technique on two different platforms and it has been proven to be consistent in decreasing the code size significantly and hence the complexity of the implementation transition system. The results illustrate that there is considerable reduction in time and effort required for the verification of a complex software control, i.e., pacemaker when statically stuttering abstracted code is employed.

1. Introduction

Today, our lives are predominantly occupied by numerous real-time embedded systems. Such systems provide specific functionality and normally they are the element of a larger system [1]. The correctness of these systems depends on the logical functions as well as on the timely response. They are used in automobiles, aircraft, implantable medical devices, cell phones, industrial robots, and many others.

At the core of embedded systems, there exist complex algorithms that define the specific working of the device. For instance, we can consider an implantable medical device like a pacemaker. Such an implanted device communicates with the patient's body and takes the real-time parameters to make decisions and execute the treatment accordingly [2]. Due to their safety-critical nature, small errors in such a system can lead to a big unrecoverable loss. Therefore, it is essential to verify the functional correctness of such systems [3]. The verification process in the phase of medical device designing can prevent the patient from unrecoverable or irreversible consequences. Likewise, the formal verification practices applied in the aircraft designing can save an ample amount of money invested in the design and prevent any unseen failures that may occur in the future due to the presence of any error. From 2006 to 2011, the US Food and Drug Administration (FDA) has reported 5294 recalls and 1,154,451 adverse events due to the malfunctioning of medical devices. 22.8% of the recalls are due to the software

bugs in the device [4]. FDA issued 55 Class 1 recalls due to software errors in the medical devices from 2006 to 2019 [5]. By looking into the statistics, it can be concluded that the formal verification of real-time embedded systems like safety-critical medical devices is indispensable.

Formal verification finds hard corner-case errors and ensures that the system is bug-free [6]. It is one of the most important and crucial phases of the software design cycle. Several efforts have been made to devise the techniques for enhancing efficiency and minimizing the complexity of the verification process. The software of an embedded system is comprised of intricate algorithms that are designed according to the required behavior of system [7]. Algorithms are written in a high-level language like C, Python, and Verilog. The compiler converts this source code into equivalent machine code, which is also known as object code. The conversion process of code from high-level language to low-level language can introduce errors in the system. Each instruction in a high-level language like C or C++ is converted into several numbers of lines in a low-level language like assembly. The object code of an embedded system is large, real-time, and interrupt-driven, and it contains the stuttering nature. The large size of the object code, which is to be executed on the embedded system, makes the application of the formal verification process more challenging. This leads to the requirement of an abstraction technique that reduces the length of the code while preserving the essence of functionality.

The abstraction technique is designed to reduce the time and effort involved in the verification process by minimizing the size of the object code. In this paper, we propose a novel abstraction technique named Static Stuttering Abstraction (SSA), which is applied to the object code statically, which is before the actual run time. The technique is designed in the context of refinement-based verification, which is a formal verification technique. The specification and implementation of the systems are denoted in the form of a transition system (TS) in refinement-based verification. The required behavior of the system is defined in specification TS through states and transitions, while the software implementation at the object code level which is executed in the embedded system is represented by the implementation TS. The size of implementation TS is very huge as compared to the specification TS. The single progression at specification TS is represented by millions of transitions in the implementation TS. These several transitions are known as stuttering transitions. Each state in the specification TS represents a unique state of the systems. The stuttering transitions arise from the execution of stuttering instructions. The states associated with the stuttering transitions in the implementation TS represent the same state of the specification TS.

We used the concept of stuttering instructions and stuttering transitions to formulate the concept of SSA. The finite number of stuttering instructions forming a pattern with a certain frequency is abstracted into one instruction. The abstracted and merged instruction preserves the functionality of the implementation TS and consequently reduces the size. In this paper, we presented a process to apply SSA on the stuttering instructions of the implementation TS. The idea of SSA was presented in the Third International Conference on Cyber-Technologies and Cyber-Systems in Athens, Greece [8].

The specific contributions of this paper include the following:

(i) The induction-based correctness proof for the SSA algorithm

(ii) Improving the efficiency of manual verification of complex medical software control based on Well-Founded Equivalence Bisimulation (WEB) refinement

(iii) Significant reduction in the state space of implementation object code for any platform

(iv) The effectiveness of SSA algorithm for static verification of object code for nondeterministic systems where implementation can take different paths in real time (based on branches)

The devised algorithm takes the instruction set architecture (ISA) and original object code as input for any platform and then automatically creates the abstracted code. We are not aware of any abstraction technique for the object code and its verification, to the best of our knowledge. With the help of the case study, we were able to achieve the proposed outcomes in terms of the verification effort improvement. Thus, SSA leads to reduced complexity in formal verification of object code in safety-critical applications.

The rest of this paper is organized as follows. The background of the basic concepts used in the rest of the paper is presented in Section 2. Section 3 details related work. The proposed abstraction technique, designed algorithm, and its correctness are described in Section 4. Section5 describes the case study and results. The verification of the pacemaker control program with SSA and the details of efficiency achieved in verification due to SSA are explained in Section 6. Conclusion and future work are discussed in Section 7.

2. Background

The specification of the system and its implementation are modeled as TS in refinement-based FV. Definition of TS is as follows [9].

Definition 1. A TS $\mathscr{M}$ is a 3-tuple $\langle S, R, L\rangle$, where S is the set of states, R is the transition relation which is the set of all state transitions, and L is a labeling function that defines what is visible at each state. A transition is of the form $\langle w, v\rangle$, where $(w, v) \in S$.

The formal specification TS MM_s represents the high-level interpretation of the required behavior of the system design. The requirements are expressed by a small set of transitions and states in MM_s. The implementation TS MM_I is obtained after MM_s is implemented on the embedded system and a high-level C code is obtained. The C code is translated into machine code, which is the object code that corresponds to MM_I. A single execution of an instruction in the object code creates a transition in MM_I. Here comes the

role of SSA abstraction since MM_s and MM_I differ largely due to the state-space size and the number of transitions. One of the major challenges in refinement-based FV techniques is matching a small set of states, i.e., MM_s, to an infinite large state space, i.e., MM_I.

The matching between the states of MM_s with a transition of the form $\langle s, u\rangle$ and MM_I with a transition of the form $\langle w, v\rangle$ is executed based on the concept of refinement map, which is defined below.

Definition 2. Let $MM_I = \langle S, R, L\rangle$, $MM_s = \langle S', R', L'\rangle$, and $(\forall_s \in S') \wedge (\forall w \in S)\ \exists r$ such that r is a function and $r(w_1 \vee w_2 \vee w_3 \vee \ldots \vee w_n) \longrightarrow (s_1 \vee s_2 \vee s_3 \ldots \vee s_n)$, where $(s_1, s_2, s_3, \ldots, s_n) \in S'$ and $(w_1, w_2, w_3, \ldots, w_n) \in S$.

A refinement map r is applied to a state in MM_I and its projects to the corresponding state in MM_s. As stated by Definition 2, refinement map can either project to the same state in the specification $r(w) = r(v) = s$, i.e., both w and v match to the same specification state s, or project to a new state in the specification $r(w) = s$ and $r(v) = u$, i.e., both w and v match to different specification states s and u, respectively. Refinement map only projects a certain set of values responsible for mapping the current state in MM_I to the corresponding state in MM_s. Therefore, if an instruction does not modify the set of values projected by the refinement map, the instruction is called a stuttering instruction s_i, as defined in Definition 3.

Definition 3. Let $I = \{i_1, i_2, i_3, \ldots, i_n\}$ be the set of instructions in MM_I, where each $i_n = \{P_n, w_n\}$, P_n corresponds to the set of values that project to the matching state in MM_s and w_n reflects the current implementation state obtained after execution of i_n; when $r(P_n)$ is applied, i_n is a stuttering instruction s_i if

(1) $i_{n-1} \wedge r(P_{n-1}) \longrightarrow s \wedge$

(2) $i_n \wedge r(P_n) \longrightarrow s \wedge$

(3) $L(w_{n-1}) = L(w_n)$

The high-level C code of a real-time system may contain common operations, which are translated into the same block of instructions in assembly code. A single block of instruction may occur multiple times, which makes a large portion of assembly code. The fundamental objective of SSA is to identify the location of common blocks of stuttering instructions s_i, consider them as a pattern, and replace that pattern with a merger of a single line instruction. The updated single line instruction includes operations of all the instructions in the replaced pattern. If a pattern consists of 4 stuttering instructions and it occurs 50 times in the object code, the SSA will reduce 150 lines in the code, thus reducing the state space of MM_I.

3. Related Work

Real-time applications like medical devices have become more efficient and flexible, which makes them more complex. This leads to an increasing demand for effective verification techniques. In 2010, the FDA launched a program "Infusion Pump Improvement Initiative" to address the problems linked with the infusion pumps malfunctioning [10]. As a result of this safety initiative, many formal verification methods for the correctness of infusion pump software were developed. These methods are based on verification tools like Kronos [11] and UPPAAL [12], which work on the concept of timed automata [13]. These methods have been successful for the verification of high-level models that work in real time. Epsilon [14] is another real-time verification tool that intends to verify the communication protocols. These tools deal with the high-level models, while we intend to do formal verification of low-level object code that is executed on the device. Our research work aims to reduce the effort required and the complexity involved in applying verification techniques by introducing an abstraction procedure. In recent research, there have been a lot of techniques based on the concept of stuttering aiming to improve the efficiency of the verification process.

Groote and Wijs in [15] provide an improved algorithm that determines the stuttering equivalence on a Kripke Structure. The time complexity of the algorithm is $O(m \log n)$, where m represents the total number of transitions and n denotes the total number of states in the structure. The equivalence determines whether the two states have unique behavior while ignoring the transitions. The authors of [16] proposed a formal verification methodology for six kinds of stepper motor control by using the Well-Founded Simulation- (WFS) based refinement. Rob [17] presented the theorems and definitions that describe how to reduce requirements for applying fair stuttering refinements for the system. Jain and Manolis presented a method to test the functional correctness of software and hardware through refinement in [18]. This method overcomes the loopholes in the de facto testing method that is usually used in the industrial sector. Rabinovich [19] used the theory of automata to stimulate continuous and discrete timed systems. The concept of stuttering is used for w-string but not for abstraction.

Stuttering is introduced by Joachim [20] for constructing the deterministic w-automata. The redundant states that are stuttering are skipped to make the size of automaton smaller. The authors employ stuttering-based reduction but did not apply abstraction on object code statically. Jabeen et al. [21] presented Timed Well-Founded Simulation (TWFS) refinement for formal verification of real-time Field Programmable Gate Array (FPGA). The authors identify the reachable states of FPGA through manually produced invariants, without using stuttering abstraction. The proposed method is suitable only for FPGA and not for the object code of a real-time system. Peter and Jose [22] introduced abstraction in the field of discrete time to reduce the size of the state space and to make analysis and model checking more feasible. The concept of abstraction is used but stuttering abstraction is not employed. Mohana et al. [23] presented a technique that automates the Well-Founded Simulation-(WFS-) based refinement on an implementation TS. The presented algorithm takes the abstracted object code as input.

The abstraction is not applied statically on object code. Shuja et al. [24] presented refinement-based verification for DDD mode pacemaker control. The object code of the pacemaker is verified by proof obligations. The authors did not consider stuttering abstraction to make the object code smaller in size, which in results makes the verification process more tedious and time-consuming.

A new abstraction framework for a 3-valued model is proposed by Nejati et al. [25]. The method avoids the use of unnecessary additional refinement step and makes the model conclusive.

Our technique aims to abstract the recurring patterns of instructions in object code. The abstraction of patterns reduces the number of transitions and makes the refinement-based verification process easier. From the results, it is evident that SSA reduces the size of the object code (MM_I). SSA algorithm and its correctness are explained in the next section.

4. Automatic Static Stuttering Abstraction

Refinement-based verification of a real-time embedded system application is complex due to the large size of the object code. In this paper, we developed an algorithm for SSA and applied that to a real-time case study. We studied that the proposed abstraction technique reduces a considerable amount of effort required for the verification of object code and elaborated results with the help of case study discussed in Section 6. SSA inputs the implementation TS MM_I (original object code) and outputs the updated MM_I (abstracted object code). With the reduction in the size of object code, the number of transitions in the updated object code also decreases. The abstraction involves the notions of stuttering and nonstuttering transitions which are formally defined as follows.

Definition 4. A stuttering transition s_t in $MM_i = \langle S, R, L\rangle$ is of the form $\langle w, v\rangle \in R \ (w, v) \in S$ if

(1) $(r(w) \longrightarrow s)\wedge$

(2) $(r(v) \longrightarrow s)\wedge$

(3) $L(w) = L(v)$

where specification TS is of the form $MM_S = \langle S\prime, R\prime, L\prime\rangle$, $\langle s, u\rangle$ is a transition in $R\prime$, and $(s, u) \in S\prime$.

Similarly, nonstuttering transition is defined as follows.

Definition 5. A nonstuttering transition n_t in $MM_I = \langle S, R, L\rangle$ is of the form $\langle w, v\rangle \in R \ (w, v) \in S$ if

(1) $(r(w) \longrightarrow s)\wedge$

(2) $(r(v) \longrightarrow u)\wedge$

(3) $L(w) \neq L(v)$

where specification TS is of the form $MM_S = \langle S\prime, R\prime, L\prime\rangle$, $\langle s, u\rangle$ is a transition in $R\prime$, and $(s, u) \in S\prime$.

The theoretical relation between stuttering instruction s_i and stuttering and nonstuttering transitions, s_t and n_t, respectively, can be recognized with the help of Definitions 3–5.

Theorem 1. *Stuttering instruction s_i only corresponds to stuttering transition s_t.*

Proof. If an instruction i_n is a stuttering instruction s_i, then there is no progress in MM_I with respect to MM_S; we apply rank function on MM_I and get $\text{rank}(w_n) < \text{rank}(w_{n-1})$ representing that the states (w_{n-1}, w_n) in MM_I both project to the same state s in MM_S. Therefore, it implies that transition $\langle w_{n-1}, w_n\rangle$ is always stuttering transition s_t if an instruction i_n is a stuttering instruction s_i. □

Rank: rank is a function that can be described with a well-founded structure comprising a set of natural numbers and a less than operator on the natural numbers ($\langle \mathbb{N}, < \rangle$). In implementation TS, it is defined as a function whose value decreases when the implementation stutters with respect to the specification, for each single stuttering transition.

Definition 6. Static Stuttering Abstraction (SSA) is an algorithm that automatically applies stuttering abstraction (SA) during static symbolic simulation of object code. The main objective of SSA is to reduce the size of the object code while preserving the actual functionality of the object code. Let $I = \langle i_1, i_2, i_3, \ldots, i_n\rangle$ be the set of instructions in the implementation TS MM_I. We can apply SA on a segment of instructions $\langle i_2, i_3, i_4, i_5\rangle$ if

(1) All the instructions in the segment are stuttering instructions s_i. If any instruction in the segment is a nonstuttering instruction, then we cannot apply SA to abstract that segment.

(2) Segments should not contain any branch/jump instruction in the middle or start of the segment.

After the application of SSA, the instructions in object code get updated with the abstracted segment in the following manner: $I = \langle i_1, i_{2\text{–}3\text{–}4\text{–}5}, i_6, \ldots, i_n\rangle$.

By using the definitions, we proposed a tool that statically applies stuttering abstractions on the object code. Algorithm 1 exhibits a procedure that performs SSA at the object code. The inputs to the procedure *Stuff_Abs* are as follows:

(1) Original object code file (obj_{init}).

(2) Matrix ($patt_{opc}$) containing information regarding the opcode of instructions that are involved in a pattern. The matrix used in our case study is given in Figure 1. Each row in the matrix depicts information about a pattern. The first column shows the mnemonics of the instructions involved in each pattern, while the rest of the columns contain the opcode of each mnemonic involved in the pattern. The first row of the matrix lists the pattern LDR-STR, which based on observation has the highest frequency in obj_{init}. The opcode of LDR is stored in the second column, while the opcode of STR is shown in the third column. This pattern contains only two instructions; the third and fourth columns have no opcode value (X). The patterns in $patt_{opc}$ are stored in the order from highest to lowest in terms of the number of

```
(1)  procedure Stutt_Abs(obj_init, pat_opc, ref − map)
(2)  counts = No − of − Rows(patt_opc)
(3)  N_c = 0
(4)  objt(N_c)⟵compute_length(obj_init)
(5)  Total_red = 0
(6)  red_l(N_c) = 0
(7)  abs_i(N_c) = 0
(8)  while N_c < count do
(9)     N_c + +
(10)    red_l(N_c) = 0
(11)    s_p(N_c)⟵patt_size(patt_opc(N_c, : ))
(12)    obj_l(N_c) = obj_l(0) − Total_red
(13)    red_p(N_c) = (s_p(N_c) − 1)
(14)    c_ins_line = 0
(15)    N_opc = 0
(16)    abs_i(N_c) = 0
(17)    while (c_ins_line < obj_l(N_c)) do
(18)    N_opc + +
(19)    red_l(N_c) = 0
(20)    c_ins_line + +
(21)    if[(N_c = 1)∧(abs_i(N_c) = 0)]then
(22)       I_c⟵Next − Ins − Fetcher(obj_init)
(23)    else
(24)       I_c⟵Next − Ins − Fetcher(obj_upd)
(25)    Match⟵Same(Opc, patt_opc(N_c, N_opc))
(26)    Opc⟵Opc − cal(I_c)
(27)    if[Match = 1]then
(28)       buff(Nope)⟵I_c
(29)       if[s_p = N_opc]then
(30)          nat⟵ref − map(buff)
(31)          N_opc = 0
(32)          if[nat = 1]then
(33)             obj_upd⟵mrg(N_c, buff, obj_init)
(34)             abs_i(N_c) + +;
(35)             red_l(N_c) = red_l(N_c) + red_p(N_c)
(36)             obj_l(N_c) = obj_l(N_c) − red_p(N_c)
(37)             c_ins_line = c_ins_line − red_p(N_c)
(38)          else
(39)             buff − initialized − again
(40)       else
(41)       N_opc = 0
(42)    End While
(43)             Total_red = ∑_{n=0}^{N_c} red_l(N_c)
(44) End While
(45)             lines_abs = obj(0) − Total_red
(46) return obj_upd
```

Algorithm 1: Procedure for Static Stuttering Abstraction.

occurrences in obj_{init}. The number of instructions in a pattern ranges from 2 to 4.

(3) A function of refinement map $(ref - map)$.

The output of the algorithm *Stutt_Abs* is an updated and reduced length of object code obj_{upd} which is the implementation TS after the abstraction (line 46).

count represents the number of total patterns in $patt_{opc}$ and it is statically computed through a function $No - Of - Rows$ (line 2). N_c keeps a track of patterns that have been abstracted so far in the algorithm. Its value in algorithm ranges from 0 (line 3) to *count*. Array obj_l stores the number of lines in object code at each iteration N_c and it is of size *count*. $obj(0)$ represents the number of lines in the initial object code obj_{init} (line 4). $Total_{red}$ represents the total number of lines reduced in obj_{init} or obj_{upd} through the process of abstraction. Its initial value is 0 (line 5). Array $abs_i(N_c)$ stores the number of times $inner - loop$ apply abstraction in object code for each pattern $patt_{opc}(N_c, ;)$. The initial value of $abs_i(N_c)$ is 0 at the start of each iteration of outer loop (lines 7 and 16) and its value increments by 1 after each abstraction (line 34). The $outer - loop$ (lines 8–44) runs the whole process of abstraction in object code for all patterns mentioned in $patt_{opc}$. It considers each

pattern stored in $patt_{opc}(N_c,;)$ during an iteration. s_p calculates the number of instructions in each pattern that is already defined in N_c row of matrix $patt_{opc}$. It is computed through a function *patt_size* (line 11). In our case study, $2 \leq s_p \leq 4$, but it must be greater than or equal to 2 in every case. Array $obj_l(N_c)$ keeps the record of updated number of lines in each iteration N_c after the possible abstraction of a pattern (line 12). Array $red_p(N_c)$ shows the number of lines that are to be reduced by the abstraction of the current pattern $patt_{opc}(N_c,:)$ (line 13). If $s_p(N_c)$ is 4, then $red_p(N_c)$ will always be 3. Variable *c_ins_line* shows the current instruction line of the object code throughout the iterations of an *inner – while* loop. Its value is restored to 0 (line 14) before *inner – while* loop gets started and goes up to the value of $obj_l(N_c)$ (line 17). Its value gets incremented by one in the beginning of each iteration of $inner - loop$ (line 20). N_{opc} variable keeps track of the number of instructions in each pattern that $inner - loop$ is locating in object code. It is initialized to 0 before $inner - loop$ (line 15) and gets incremented at the start of each iteration (line 18).

$inner - loop$ scans the whole object code for each pattern $patt_{opc}(N_c,:)$ (lines 17–42). It conducts the process of locating and abstracting each pattern in complete object code. The next instruction I_c in obj_{init} or obj_{upd} is obtained through a function $Next - Ins - Fetcher$. obj_{init} is considered if $outer - loop$ is running for the first pattern ($N_c = 1$) and until now there is no abstraction conducted on the object code ($abs_i(N_c) = 0$); otherwise obj_{upd} is considered (lines 21–24). Opc represents the opcode of I_c and is calculated through a function $Opc - Cal$ (line 25). For locating a pattern in object code, Opc of I_c must be equal to already defined opcode of an instruction in $patt_{opc}(N_c, N_{opc})$. Opc and $patt_{opc}(N_c, N_{opc})$ are compared through function *Same*. The variable will be 1 if both opcodes are equal (line 26). If ($Match = 1$) (line 27), then I_c is stored in $buff$ (line 28); else N_{opc} will be set to 0 (line 41). N_{opc} must be equal to s_p, for the abstraction of instructions stored in $buff$ (line 28). It indicates that required number of instructions in a pattern $patt_{opc}(N_c,:)$ is located by the procedure in object code and is stored in $buff$. Another condition for the abstraction of instructions stored in $buff$ is that they all should be a stuttering instruction s_i. The stuttering or nonstuttering nature of instructions is computed using a function $ref - map$ (line 30). $ref - map$ will return 1 in the case where all the instructions in $buff$ are stuttering instructions; otherwise, output will be 0. If instructions in $buff$ are stuttering instructions s_i (line 32), the pattern in object code is abstracted through a function mrg. This function returns the updated object code obj_{upd} (line 33). $red_l(N_c)$ is an array that stores the number of lines reduced after the abstraction of each pattern $patt_{opc}(Nc,:)$ in an object code (line 35). The initial value of $red_l(N_c)$ for each iteration N_c is 0 (lines 6 and 10). In obj_{upd}, the total number of lines $obj_l(N_c)$ and current line *c_ins_line* get reduced by $red_p(N_c)$ (lines 36 and 37). N_c will be incremented by one when the search for a pattern ends in entire object code (line 9). It indicates that current inner loop (lines 13–37) will search for a pattern in $patt_{opc}(N_c)$.

$Total_{red}$ updates the data of total number of lines reduced red_l after each execution of inner loop, abstracting each pattern $patt_{ope}(N_c)$ (line 43). *lines_abs* is the total number of lines in object code after the complete abstraction process (line 45).

obj_{upd} is smaller in size but it contains the original functionality of theobj_{init}. The prime functionality of the SSA algorithm is to reduce the size of the object code while preserving the essence of the original object code. Figure 2 gives a graphical overview of the SSA tool working with a segment of instructions on which a single merger is applied.

The instructions in the merger are from the pacemaker object code (obj_{init}) implemented on an embedded system platform LPC1768 and intend to do following operations:

(1) Load the value at a memory location [0 × 000007D0] to a register r1

(2) Store the value in register r0 at memory location addressed by [[0 × 000007D0] + #0 × 00]

$Buffer$ includes a pattern (LDR-STR) that is defined in the matrix $patt_{opc}(1,:)$ in Figure 1. For the abstraction, the instructions in a buffer must be stuttering instructions. The state of the system for pacemaker object code is associated with the memory address 0 × 00000820. The above pattern is not changing the state of the system and operation in both instructions cannot be executed by the microprocessor in a single execution cycle. We replace the instructions in $Buffer$ with a $Merger$. The merger identifier is named *LST*. The merger is updating the content of memory location [0 × 000007D0 + #0 × 00] with the value of register r0. The opcode of the merger is updated by the ASCII conversion of the merger. The instructions in buffer occupy memory address from 0 × 00000642 to 0 × 00000644 but the merger is occupying a single memory location. The LDR-STR pattern occurs 35 times in obj_{init} of a pacemaker as shown in Table 1. The abstracted object code reduces the length of object code and the number of stuttering transitions s_t in MM_I. The reduced length of the object code makes the verification process easier for the end user. The information regarding patterns, their mergers, and opcode is provided in Table 1.

4.1. Correctness of SSA. In this section, we presented the correctness of the SSA algorithm to ensure its correct functionality. The correctness of an algorithm is stated when an algorithm is correct according to its specification. If an algorithm gives an expected output for input, then it refers to its functional correctness [26].

The total correctness of an algorithm requires its partial correctness and termination proof [27]. The partial correctness of an algorithm ensures that whenever algorithm runs it always outputs the correct value [28]. In Algorithm 1, after a few assignments, a large section (lines 8–44) is repeated to achieve the desired output; therefore, it is an iterative algorithm [29]. The correctness of Algorithm 1 is completely dependent on the correctness of the loop (lines

8–44). The partial correctness of the loop is asserted if and only if variables in algorithm satisfy the loop precondition and after the commands in the loop complete their execution, the variables of algorithm hold the postcondition [30].

Algorithm 1 inputs the object code of a medical device obj_{init}, which is the initial state, and outputs the updated object code obj_{upd} with all possible mergers of patterns, which is the final state. Initial and final states can be represented by the predicates. A precondition for an algorithm is the predicate that must be true before the algorithm starts its execution. Similarly, the predicate that states what must be true after the execution of an algorithm for a given precondition is the postcondition [31]. The partial correctness of a loop involves finding the loop invariant and then proves it through the induction method. A predicate that is true before the first iteration of a loop and is also true after the finite iterations of that loop is known as loop invariant [32]. The truth of loop invariant ensures the truth of the postcondition of the loop. We have considered that all the variables used in Algorithm 1 are natural numbers. In termination proof of an iterative algorithm, the iteration number is associated with the decreasing sequence of natural numbers. Then the termination proof can be concluded from Theorem 2 [33], given as follows.

Theorem 2. *Every decreasing sequence of natural number is finite.*

The total correctness proof can be expressed as

$$\text{precondition} \wedge \text{termination} \longrightarrow \text{postcondition}, \quad (1)$$

All the variables used in Algorithm 1 are natural numbers. Algorithm contains nested loops (inner loop (lines 17–42) and outer loop (lines 8–44)). Partial correctness of nested loops first proves the loop invariant of the inner loop and afterward for the outer loop. Sections4-A1 and 4-A2 present the total correctness of the inner and outer loop of Algorithm 1, respectively, involving its partial correctness and termination proof. The naming convention of variables used in this section is the same as that used in Algorithm 1.

4.1.1. Correctness of Inner Loop. This section describes the correctness of *inner − loop* (lines 17-42) of Algorithm 1. The inner loop scans entire object code and applies merger LST on all possible occurrences of each type of pattern (e.g., LDR-STR).

The correctness proof is as follows:

(1) The precondition for the inner loop is

$$\left[\left(obj_l(N_c) \geq 0\right) \wedge \left(red_p(N_c) > 0\right) \wedge \left(red_l(N_c) = 0\right) \wedge \left(abs_i(N_c) = 0\right)\right]. \quad (2)$$

It consists of 4 conditions that must be true before the execution of inner loop:

The number of lines in object code at iteration N_c of the outer loop is stored in $obj_l(N_c)$ and its value cannot be negative before the inner loop (line 12).
The number of lines that can be reduced $red_p(N_c)$ due to each abstraction of a pattern $patt_{opc}(N_c, :)$ cannot be a negative value (line 13). For a two-line pattern like LDR-STR, only one line can be reduced in object code after applying merger LST.
The total number of lines reduced $red_l(N_c)$ through abstraction of a pattern $patt_{opc}(N_c, :)$ in whole object code must be zero (line 10).
The number of times abstraction $abs_i(N_c)$ of a pattern $patt_{opc}(N_c, :)$ is applied on object code must be zero.

(2) The postcondition for inner loop is

$$red_l(N_c) = red_l(N_c) * abs_i(N_c). \quad (3)$$

It must be true after the termination of inner loop, when the condition $c\_ins\_line = obj(N_c)$ becomes true. The purpose of inner loop is to apply merger on all possible occurrences of a pattern $patt_{opc}(N_c, :)$ in object code. The inner loop calculates information regarding total number of lines reduced in object code $red_l(N_c)$ and total number of times merger is applied on object code $abs_i(N_c)$ for a pattern given in matrix $patt_{opc}(N_c, :)$.

(3) The loop invariant is stated as

$$I(n): red_l(N_c) = red_p(N_c) * abs_i(N_c). \quad (4)$$

$red_p(N_c)$ is a constant value for a complete run of inner loop. For convenience, we considered $red_p(N_c)$. We will now prove the loop invariant by the induction method on the iteration number of the loop. k denotes the iteration number of the loop. For iteration k, the loop invariant is

$$I(n): red_{l_k}(N_c) = const * abs_{i_k(N_c)}. \quad (5)$$

Proof. The **base case** for loop is $k = 0$. Before the loop starts (line 17), $I(0)$ is

$$\begin{aligned} red_{l_0}(N_c) &= const * abs_{i_0}(N_c) \\ &= (const * 0) \\ &= 0. \end{aligned} \quad (6)$$

This indicates that loop invariant holds for the base case. $abs_{i_0}(N_c)$ represents the initial value of $abs_i(N_c)$. It is always 0 for every N_c before the inner loop is executed (line 16).

$$\text{patt}_{\text{opc}} = \begin{bmatrix} \text{LDR (pc), STR} & 01001 & 01100 & X & X \\ \text{MOV (32-bit), STR} & 1111000001001111 & 01100 & X & X \\ \text{LSLS, STR} & 00000 & 01100 & X & X \\ \text{MOVS, MOV (32-bit), STR} & 00100 & 1111000001001111 & 01100 & X \\ \text{LDR (pc), LDR (Register), CMP, BNE} & 01001 & 01101 & 01101 & 11010 \\ \text{LDR (pc), LDR (Register)} & 01001 & 01101 & X & X \\ \text{TLR (Merger), STR} & 100000100 & 01100 & X & X \\ \text{LDR (pc), STR (32-bit)} & 01001 & 1111100000000001 & X & X \\ \text{LST (Merger), TLR (Merger)} & 01110110 & 100000100 & X & X \\ \text{MOV (32-bit), LDR (Register)} & 1111000001001111 & 01101 & X & X \\ \text{LST (Merger), LST (Merger)} & 01110110 & 01110110 & X & X \\ \text{OMS (Merger), OMS (Merger)} & 01111001 & 01111001 & X & X \\ \text{TLR (Merger), CBNZ} & 100000100 & 10111 & X & X \\ \text{OMS (Merger), LST (Merger)} & 01111001 & 01110110 & X & X \end{bmatrix}$$

Figure 1: Pattern opcode matrix.

If the loop invariant I is true for k^{th} iteration, then it should hold for $(k+1)$ iteration (**induction step**). From lines 34 and 35 of algorithm,

$$\begin{aligned} abs_{i_{(k+1)}}(N_c) &= abs_{i_k}(N_c) + 1, \\ red_{l_{(k+1)}}(N_c) &= red_{l_k}(N_c) + red_p(N_c) \\ &= red_{l_k}(N_c) + const, \end{aligned} \tag{7}$$

From equation (2),

$$\begin{aligned} red_{l_{(k+1)}}(N_0) &= \left[const * abs_{i_k}(N_c)\right] + const \\ &= const * \left[abs_{i_k}(N_c) + 1\right], \end{aligned} \tag{8}$$

From equation (4),

$$= const * abs_{i_{(k+1)}}(N_c). \tag{9}$$

From equation (5), it can be seen that loop invariant holds after the $(k+1)^{th}$ iteration. By induction method, the truth of equations (2) and (5) shows that loop invariant (equation (1)) is also true.

(4) The **termination proof** for inner loop is given as follows. □

Proof. The guard **G** of the loop (line 17) is

$$c\_ins\_line < obj_l(N_c). \tag{10}$$

The algorithm shows that $obj_l(N_c)$ is a constant (line 12) or a decreasing value (line 36). From algorithm lines 36 and 37, we can see that, after abstraction,

$$\begin{aligned} obj_l\left(N_c = obj_l(n_c) - red_p(N_c)\right), \\ c\_ins\_line = c\_ins\_line - res_p(N_c). \end{aligned} \tag{11}$$

We only considered the constant nature of $obj_l(N_c)$, as the decrease in $obj_l(N_c)$ is nullified by the decrease in *c_ins_line*.

The initial value of *c_ins_line* is 0 (line 14). Due to increment in *c_ins_line* during each iteration of inner loop (line 20) its value gets closer to $obj_l(N_c)$. Suppose that d_k denotes the difference between total number of lines in object code $obj_l(N_c)$ and current instruction line of object code at $c\_ins\_line_k$. k denotes the iteration number. From equation (6), d_k can be written as

$$d_k = \left(obj_l(N_c) - c\_ins\_line_k\right) \geq 0. \tag{12}$$

d_k belongs to set of natural numbers N. We can see from line 20 of algorithm that

$$c\_ins\_line_{(k+1)} = c\_ins\_line_k + 1. \tag{13}$$

After the $(k+1)^{th}$ iteration, equation (7) will become

$$\begin{aligned} d_{(k+1)} &= obj_l(N_c) - c\_ins\_line_{(k+1)} \\ &= obj_l(N_c) - \left(c\_ins\_line_k + 1\right) \\ &= \left(obj_l(N_c) - c\_ins\_line_k\right) - 1 \\ &= \left(d_k - 1\right) < d_k. \end{aligned} \tag{14}$$

Equation (8) shows that d is a decreasing sequence of iteration k. From Theorem 2, we know that loop terminates in finite steps.

(5) The truth of postcondition of inner loop can also be validated through the results given in Table 1. Consider a 4-line pattern LDR (pc), LDR (Register), CMP, BNE given in Table 1.

The postcondition is given as

$$\begin{aligned} red_l(N_c) &= red_p(N_c) * abs_i(N_c) \\ &= \left(s_p(N_c) - 1\right) * abs_i(N_c) \\ &= (3 * 13) \\ &= 39. \end{aligned} \tag{15}$$

The computed value can be compared with the value of $red_l(N_c)$ given in Table 1. □

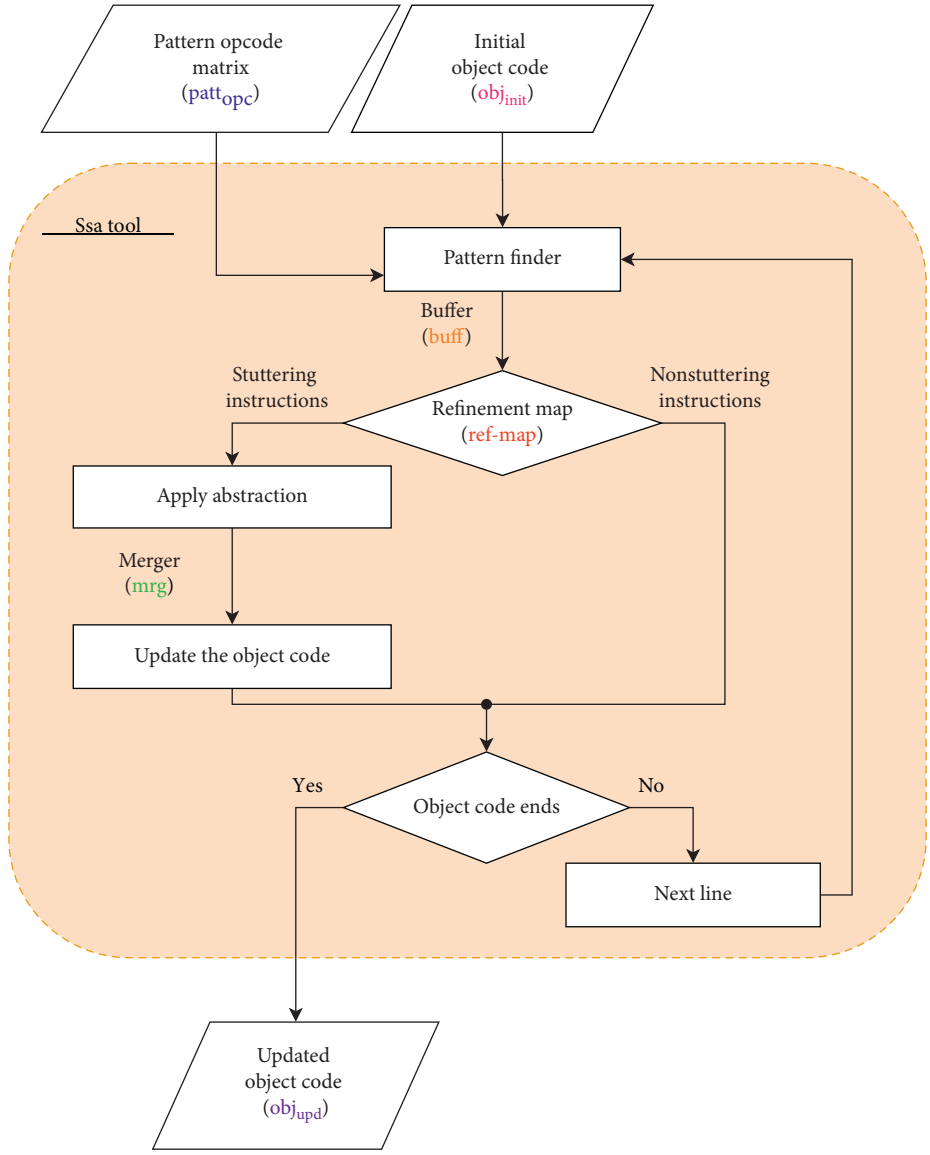

Figure 2: Graphical overview of the SSA tool.

4.1.2. Correctness of Outer Loop. This section proves the correctness of the outer loop (lines 8–44) of the SSA algorithm. The main purpose of the outer loop is to consider each pattern from the matrix $patt_{opc}$ during each iteration and calculate the total number of reductions $Total_{red}$ (line 43):

(1) The **precondition** for the outer loop is

$$[(count \geq 0) \wedge (Total_{red} = 0) \wedge (N_c = 0) \wedge (red_l(N_c) = red(0) = 0) \wedge (abs_i(N_c) = abs_i(0) = 0)]. \quad (16)$$

The precondition consists of 5 conditions, which must be true before the outer loop execution.

count represents the total number of patterns (rows) in the matrix $patt_{opc}$ (line 2). Its value cannot be negative.

$Total_{red}$ represents the total lines reduced due to the abstraction of patterns in object code (line 43). Its value must be zero as the precondition of $outer - loop$.

Table 1: Information on patterns and mergers of LPC1768 object code for pacemaker.

Instruction type (N_C)	No. of lines in pattern (Sp)	Frequency of pattern (pacemaker) abs_i (N_c)	No. of lines reduced (pacemaker) red_l (N_c)	Instruction opcode (Opc)	Abstracted merger label M_{erger}	Merger opcode (ASCII)	Merger opcode (binary)
$LDR_{(pc)}$, STR	2	35	35	[01001], [01100]	LST	768384	01110110
MOVS, STR	2	12	12	[00100], [01100]	MST	778384	1110111
LSLS, STR	2	0	0	[00000], [01100]	STL	838476	10000110
MOVS, LST (user-defined)	2	22	22	[00100], [1110110]	OMS	797783	1111001
MOVS MOV (32-bit), STR	3	8	8	[00100], [F04F], [01100]	VMS	867783	10000110
LDR (pc), LDR (register), CMP, BNE	4	13	39	[01001], [00100], [00101], [11010001]	BCL	666776	1100110
LDR (pc), LDR (register)	2	15	15	[01001], [00100]	TLR	847682	10000100
LDR (pc), LDR (register), STR	2	3	3	[01001], [00100], [01100]	RLS	828376	10000010
LDR (pc), STR (32 bit)	2	7	7	[01001], [F8C1]	DLS	687683	1101000
LST (user-defined) TLR (user-defined)	2	2	2	[01110110], [10000100]	NLT	787684	1111000
MOV (32-bit) LDR (register)	2	3	3	[F04F], [00100]	CML	677776	1111000
LST (user-defined) LST (user-defined)	2	0	0	[01110110], [01110110]	ELT	697684	1101001
OMS (user-defined) OMS (user-defined)	2	2	2	[01111001], [01111001]	FOS	707983	1110000
TLR (user-defined) CBNZ	2	2	2	[10000100], [10111]	ZTR	908482	10010000
OMS (user-defined) LST (user-defined)	2	0	0	[01111001], [01110110]	MLO	777679	01110111

N_c indicates the current pattern (row number) considered in the matrix $patt_{opc}$. Before the outer loop, its value must be zero.
$red_l(N_c)$ represents the total number of lines reduced in object code due to the abstraction of pattern $patt_{opc}(N_c, :)$. Its value must be zero before the process of abstraction (outer loop) starts when $N_c = 0$.
$abs_i(N_c)$ shows the total abstractions of a pattern $patt_{opc}(N_c, :)$ done by the inner loop. Before the outer loop ($N_c = 0$), its value must be zero.

(2) The **postcondition** of outer loop is

$$Total_{red} = \sum_{n=0}^{N_c} red_l(N_c). \tag{17}$$

By putting value of $red_l(N_c)$ from equation (1),

$$Total_{red} = \sum_{n=0}^{N_c} \left[abs_i(n) * red_p(n)\right]. \tag{18}$$

The postcondition is returned by the loop when guard of the loop becomes false. The guard **G** of the loop is $N_c = count$. The main purpose of outer loop is to calculate the total number of lines reduced *line_abs* (line 45) in object code due to the abstraction process conducted in inner loop.

(3) **The loop invariant** for outer loop is

$$I(n): Total_{red} = \sum_{n=0}^{N_c} \left[abs_i(n) * red_p(n)\right]. \tag{19}$$

The **partial** correctness of $outer - loop$ is done by proving the loop invariant (equation 11) through induction method on number of iterations k. For iteration k, the loop invariant is

$$I(n_k): Total_{red_k} = \sum_{n_k=0}^{N_{c_k}} \left[abs_i(n_k) * red_p(n_k)\right]. \tag{20}$$

Proof. Suppose that the base case for loop is $k = 0$ and $n_k = 0$.

Before the first iteration of loop starts (line 8), the loop invariant in equation (12) becomes

$$I(0): Total_{red_0} = abs_i(0) * red_p(0) = 0 * red_p(0) = 0. \tag{21}$$

$abs_i(0)$ is the precondition of the outer loop. The loop invariant is true for the base case (k).

Induction step: in the iteration $(k+1)$, the loop invariant $I(n_{(k+1)})$ will be

$$Total_{red_{(k+1)}} = Total_{red_k} + \left[abs_i\left(n_{(k+1)}\right) * red_p\left(n_{(k+1)}\right)\right]. \tag{22}$$

By putting value of $Total_{red_k}$ from equation (12), equation (13) will become

$$\begin{aligned} I\left(n_{(k+1)}\right): Total_{red_{(k+1)}} &= \sum_{n_k=0}^{N_{c_k}} \left[abs_i(n_k) * red_p(n_k)\right] + abs_i\left(n_{(k+1)}\right) * red_p\left(n_{(k+1)}\right) \\ &= \sum_{n_k=0}^{N_{c_k}} \left[abs_i\left(n_{(k+1)}\right) * red_p\left(n_{(k+1)}\right)\right]. \end{aligned} \tag{23}$$

Comparison of equations (12) and (14) shows that the loop invariant $I(n)$ holds after the $(k+1)^{th}$ iteration of the loop.

(4) The **termination proof** for outer loop is given as follows. □

Proof. The guard **G** of the loop (line 8) is

$$N_c < count. \tag{24}$$

From line 2, it is evident that *count* is always a constant value. The value of N_c is zero before the outer loop (line 3), but after each iteration of outer loop, its value gets closer to *count* (line 9).

Let

$$d_k = \left(count - N_{c_k}\right). \tag{25}$$

Meanwhile, $d_k > 0$ and $d_k \in \mathrm{N}$.

During $(k+1)^{th}$ iteration of the outer loop, equation (15) will become

$$\begin{aligned} d_k &= \left(count - N_{c_{(k+1)}}\right) \\ &= count - \left(N_{c_k} + 1\right) \\ &= \left(count - N_{c_k}\right) - 1 \\ &= d_k - 1 < d_k. \end{aligned} \tag{26}$$

The above equation (16) shows that the value of d_k is a decreasing sequence of positive integers. From Theorem 2, it can be concluded that the outer loop terminates after finite iterations.

The outer loop gives information regarding $Total_{red_k}$ upon its termination when $N_c \nless count$. $Total_{red_k}$ given in Table 2 is the sum of all values in column $red_l(N_c)$ given in Table 1. □

5. Case Study and Results

We have checked the efficiency of SSA abstraction tool by implementing it on the object code of a pacemaker. The code of pacemaker is designed according to the clinical settings expressed as a TS presented in Figure 3. At the start, the implementation transition system MM_I of pacemaker is obtained on an ARM Cortex-M3-based NXP LPC1768 microcontroller. A library is maintained according to the instruction set architecture (ISA) of the microcontroller. Table 1 summarizes the information about the number and type of patterns identified and merged by an automated SSA tool for the object code. SSA reduces 41.78% of the object code of the pacemaker.

To confirm the consistency of the proposed tool, the functionality of the pacemaker is also modeled on a different platform, ATMega328p microcontroller. A separate library is maintained which supports the ISA of the Atmega328p microcontroller. Table 2 shows that, after abstraction through the SSA tool, the object code of pacemaker implemented on Atmega328p microcontroller is reduced up to 23.47%. The reduction in both cases is for a single execution of code, while in the real-time application the object code is executed in an infinite execution loop. The SSA reduces the considerable number of s_t which makes the refinement-based verification process more efficient.

Figure 4 depicts the general methodology for verification with SSA. The implementation code is first reduced by using the proposed SSA algorithm and then the reduced code and specification are formally modeled as transition systems (TS). Using the theory of WEB, which incorporates the

Table 2: Results obtained on LPC1768 and ATmega328p object code for pacemaker.

Metrics	Pacemaker	
	LPC1768	*ATmega328p*
Number of lines in original object code	359	230
Number of lines reduced after the application of abstraction	150	54
Updated number of lines in abstracted object code	209	176
Total number of patterns that are detected and abstraction during SSA	15	22
Percentage of object code abstraction	41.78%	23.47%

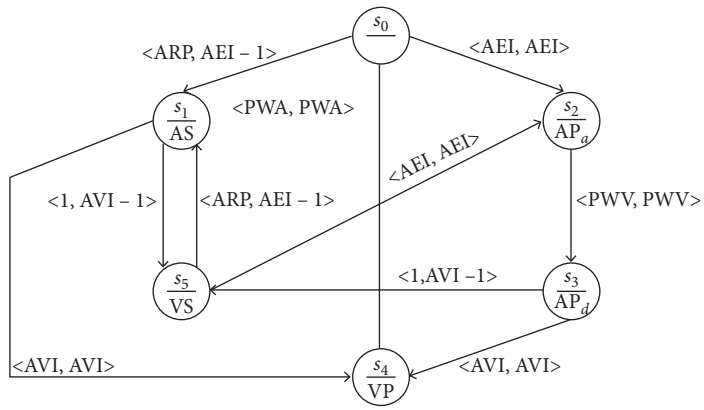

Figure 3: Specification TS of pacemaker $STS_{PM} = (S_c, R, L)$.

stuttering and nonstuttering behaviors, formal models and theorems are generated and discharged into theorem prover Z3. If the theorem is incorrect, both the formal model and theorem are revised, and the process is repeated. Previously, Shuja et al. [24] similarly verified the object code but they did not apply the SSA technique. Our case study results conclude that SSA has reduced the verification time of object code.

6. Verification of Pacemaker with SSA

To validate the correctness of abstracted object code after the SSA application, we performed the formal verification of the pacemaker control program using a statically abstracted implementation transition system (TS). It is to be noted that now the statically abstracted object code contributes to the implementation transition system instead of the original source code. We have taken the specification of pacemaker from [24] in the form of a TS (STS_{PM}) as shown in Figure 3.

The pacemaker control program was implemented on Cortex-M3-based NXP LPC1768 microcontroller. The object code obtained after implementation was abstracted using the proposed SSA tool. The resultant abstracted code was formally verified against the specification. The underlying theory used for verification is the WEB (Well-Founded Equivalence Bisimulation) refinement theory [9] which is a notion of equivalence between the two transition systems (TS). The abstracted object code which is the implementation can be modeled as a TS where both the nonstuttering instructions and abstracted stuttering instructions are modeled as functions. The object code implementation is at the lower level with greater details as compared to the specification which is only the higher-level representation of system behavior. To overcome the differences in the implementation states and specification states, WEB employs the concept of refinement maps. A refinement map is a function that projects the implementation states to specification states and its definition is given in Section 2. The number of transitions in implementation TS is far more than the number of transitions in specification TS. For our pacemaker case study, the object code or implementation has millions of transitions, while the specification has only 10 transitions. Therefore, several implementation transitions match to a single transition in the specification. This phenomenon is known as stuttering and WEB refinement theory covers this concept. A more detailed description of WEB refinement can be found in [22]. To ensure that the implementation is making progress and stuttering phenomenon terminates eventually, we have used rank functions. Rank functions discriminate against the stutter from deadlock (infinite stutter). If there is an infinite stutter, then it signals to a deadlock bug in the implementation.

We developed different proof obligations for pacemaker control knowing the specification TS. The proof obligations are written as the decidable fragments of first-order logic and are checked using the decision procedure (SMT solver Z3). The specification and implementation are encoded in the language of SMT. The proof obligations cover all the possible transitions and behaviors in statically abstracted implementation TS and check that every possible transition in the implementation system matches to a transition in the specification.

The proof obligations should encompass only the reachable states of the implementation to avoid spurious counterexamples. We derived an invariant property that

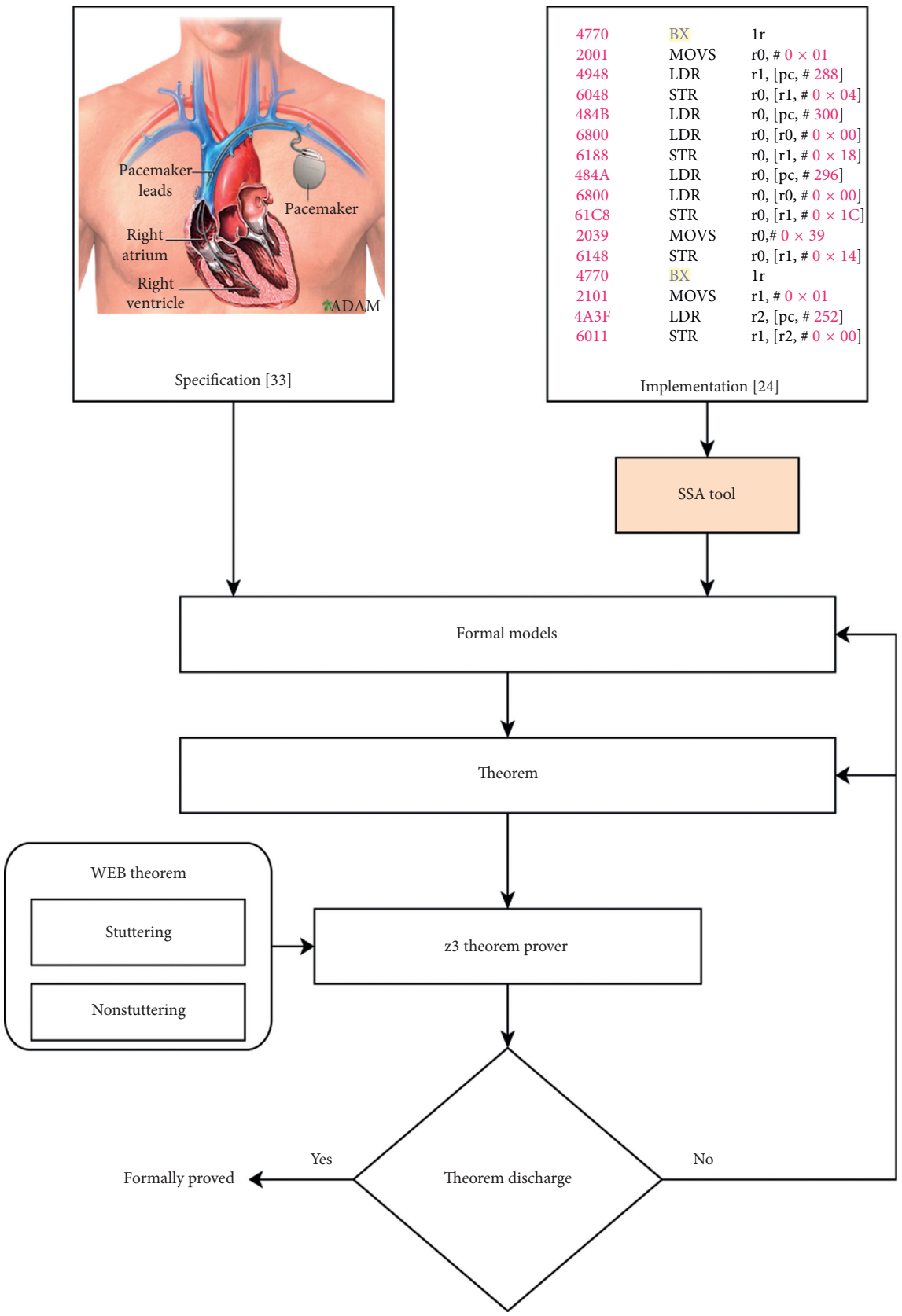

Figure 4: Verification methodology with SSA.

identifies only the reachable states of implementation. The invariant property is given as follows:

$$\begin{aligned}
&\{(r(w) = s_0) \vee (r(w) = s_5) \wedge (Vt_1 \leq AEI)\} \\
&\vee \{(r(w) = s_0) \wedge (At_1 = Vt_1)\} \\
&\vee \{(r(w) = s_2) \wedge (Vt_1 \leq PWV)\} \\
&\vee \{(r(w) = s_2) \wedge (Vt_1 \geq AEI)\} \\
&\vee \{(r(w) = s_4) \wedge (At_1 \leq PWA)\} \\
&\vee \{(r(w) = s_1) \vee (r(w) = s_3) \wedge (At_1 \leq AVI)\}.
\end{aligned} \tag{27}$$

In the abovementioned properties, w is the implementation state and r is the refinement map which projects each implementation state w to one of the specification states S_c. The invariant captures all the reachable states in the implementation. The object control program requires two counters indicated by At_l and Vt_l which keep track of the time that has passed since the last atrial and ventricle pace, respectively. The two counters show the permissible range of time based on which the transitions take place between different implementation states. These permitted ranges of time values correspond to timing cycles in the specification (STS_{PM}) and are given using constants AEI, AVI, PWV, and PWA. The invariant holds true at each and every state of the implementation system, thus ensuring that the implementation always corresponds to the specification. We derived proof obligations for stuttering and nonstuttering behaviors of implementation. The invariant property characterizes all the implementation states to ensure that every transition in the implementation matches to a transition of the specification. The developed WEB refinement proof obligations were verified by Z3. The verification experiments were performed on Intel (R) Xeon (R) Bronze 3104CPU @1.70 GHz with 64GB RAM. A verification check was performed on the updated object code using the Z3 solver.

6.1. Bugs Caught by WEB. WEB refinement-based proof obligations facilitated finding out the bugs that are otherwise unseen or neglected by testing-based approaches. We caught up to two such kinds of bugs while verifying the pacemaker's abstracted object code using proof obligations. The description of such bugs is given as follows:

(1) Bug 1: Port1 of LPC1768 was used for AP and VP, as given in the specification STS_{PM}. The I/O functionality of Port1 is controlled by *FIO1SET* and *FIO*1*CLR* registers, through which the pin values are set and cleared, respectively. The value of FIO1SET was being updated inaccurately by the object code, specifically resulting in the system to transition incorrectly from s_3 to s_0 to s_4. The correct transition according to specification STS_{PM} is from s_3 to s_4 directly. This bug was caught by WEB proof obligations as this transition (s_3, s_0) did not match to the specification transition system.

(2) Bug 2: the inputs for AS and VS were implemented using external interrupts. The interrupt status register IO2IntStatR holds the current status of interrupts. A value of either 1 or 2 in IO2IntStatR indicates that an AS or VS has occurred, respectively. The bug was revealed by VS input following the AS input and hence changing the status register IO2IntStatR value from 1 to 3. This is an incorrect value of status register indicating that both an AS and a VS have occurred. Consequently, the source of the external interrupt was misread as AS instead of VS. The bug occurred because the interrupt status was not cleared in the IO2IntStatR after the occurrence of an AS. This bug was caught and fixed.

6.2. Induced Bugs. In order to assure that SSA does not alter or hide the original behavior of the object code, we also induced some bugs in the original object code. We added two more bugs, one in nonstuttering instructions and another in stuttering instructions, to check how SSA behaves to erroneous code. We were able to catch these induced bugs also in the abstracted code. The description of the induced bugs is given as follows:

(1) Bug 1: in the pacemaker object code, the state of the system is represented by the last 5 bits [34] of *LPC_GPIO*1. First, the pins of *LPC_GPIO*1 need to be configured as output by writing 1s to the respective pins of the *FIO1DIR* register. As shown in Figure 5(a), the merger *MST* preserves the original instructions data. With the help of proof obligation, the merger is verified to implement correct behavior. Next, we induced an error by sending the wrong data to one pin of the *FIO1DIR* register. This erroneous data was passed to the merger as well, and the SSA tool abstracted the code (shown in Figure 5(b)). When this merger with erroneous data was verified, the proof was not satisfied. Hence, SSA is shown to preserve the original code functionality. Hence the encoded WEB-refinement proof obligations can catch bugs in the abstracted code.

(2) Bug 2: we induced another bug in the nonstuttering transition. As shown in Figure 6(a), the instruction STR is writing to the register *FIO1SET* at the address computed by $[r1, \#0 \times 38]$. *FIO1SET* and *FIO1CLR* are the two registers that directly change the state of the system with respect to the specification. The instructions that write to these two registers are, therefore, the nonstuttering instructions. We moved incorrect data to *FIO1SET*, which changed the system state from s_0 to s_3 (as shown in Figure 6(b)). This transition is not given in specification TS (STS_{PM} in Figure 3); therefore, it was a bug in the implementation. The SSA tool does not abstract these nonstuttering instructions preserving the original code behavior. The proof obligations for these two instructions caught the error and counterexamples were generated. With the help of counterexamples, we can fix the bugs in the implementation.

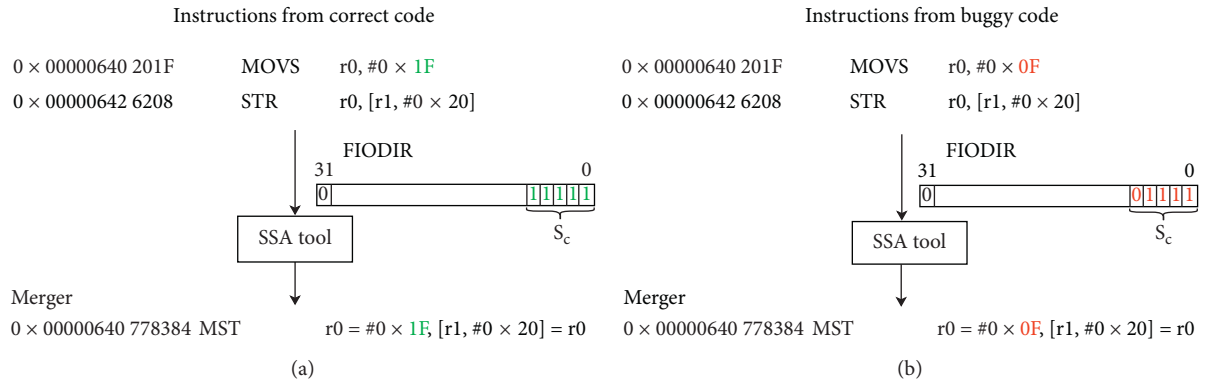

Figure 5: Induced bug in a pattern of stuttering instructions. (a) Behavior of correct code. (b) Behavior of buggy code.

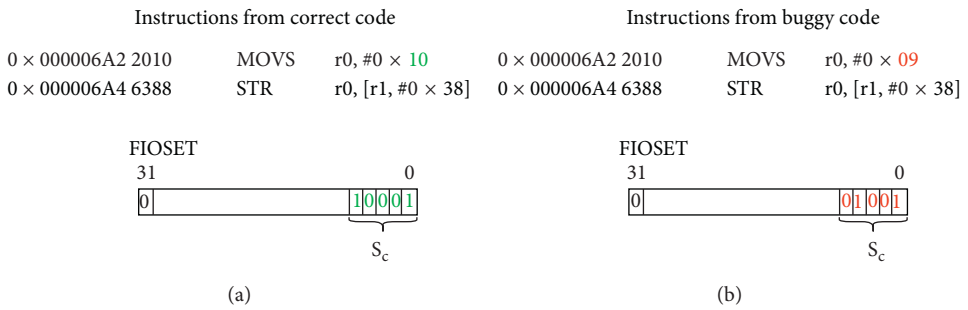

Figure 6: Induced bug in a nonstuttering transition. (a) Behavior of correct code. (b) Behavior of buggy code.

7. Verification Effort Improvement

In refinement-based verification, it is required to write the proof for each assembly instruction. When SSA is applied, the numbers of assembly instructions are reduced and, consequently, the verification effort is reduced. We describe the verification of a piece of assembly code for our case study. Let us consider a three-line code fragment from the original object program as shown in Figure 7.

The SSA tool abstracts this code fragment into a single merger and replaces every occurrence of this multiline code in the original program by a single merger. This merger actually preserves the original function of code and, in the context of verification effort, it reduces the redundancy in proving the correctness of stuttering instructions. So, in this case, instead of generating three proof files, only one file is needed to be generated. The *MOVS* instruction updates register r0 with a constant value; *LDR* instruction adds a constant to PC value and loads the value placed at that address to the r1 register. Finally, the *STR* instruction stores the content of register r0 into memory location computed by $[[0 \times 000007E4] + 0x00]$. The SSA tool merges this operation into a single instruction **OMS**, which combines the functionality of original set of instructions, eventually producing the same outcome. The proof files are written for the abstracted object code including merged (stuttering) and nonmerged instructions. Due to the space constraints, we are presenting the proof obligation for only the above abstracted instructions as given below. The proof obligations for all generated mergers are written in a similar manner with their own specific conditions:

$$\begin{aligned} \mathbf{P}0\text{:}\ \Big\{ & (r(w) = S_C) \wedge (w.r0 = prevalue) \wedge (w.pc = prevalue) \\ & \wedge (c.FIOSET = 0) \wedge (c.1 = c.value) \\ & \longrightarrow (r(v) = S_C) \wedge (rank(v) < rank(w)) \Big\}, \end{aligned} \tag{28}$$

where $S_C = V_5^{c=0} s_c$ in the specification STS_{PM}. w is the current implementation state and v is its successor. The precondition checks the current values of peripheral registers and postcondition checks that the implementation is stuttering. This proof obligation checks the correctness of merger instruction bypassing the redundant verification checks. For all other mergers in the SSA abstracted code, the same method is employed to write the proof obligations. We have developed proof obligations for all the reachable states of implementation using case analysis. The proof obligations for nonstuttering transitions are similar to those developed in [24].

Another noticeable fact is that the state of implementation system maps to the same specification state before and after the merger instruction. Therefore, the proof obligations for mergers also back the concept that mergers only combine the stuttering instructions.

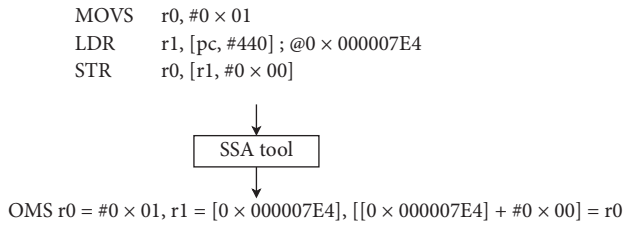

Figure 7: OMS merger.

Table 3: Comparison of transitions (in millions) in different paths (branches) of object code with and without SSA.

Path 1						Path 2							
$s0$		$s0 \to s5$		$s0 \to s5$		$s0$		$s0 \to s1$		$s0 \to s4$		$s4 \to s0$	
External interrupt at P2.0				External interrupt at P2.1		External interrupt at P2.0				No external interrupt at P2.1			
0 < timer < arp		timer = arp		arp < timer < avi		0 < timer < arp		timer = arp		timer = avi		timer = PWA	
180	91	154	78	60	30	180	91	154	78	59	25	250	170

Path 3								Path 4									
$s0$		$s2 \to s2$		$s2 \to s3$		$s3 \to s5$		$s0$		$s0 \to s2$		$s2 \to s3$		$s3 \to s4$		$s3 \to s4$	
No external interrupt at P2.0				External interrupt at P2.1				No external interrupt at P2.0						No external interrupt at P2.1			
0 < timer < aei		timer = aei		timer = PWV		aei < timer < avi		0 < timer < aei		timer = aei		timer = PWV		timer = avi		timer = PWA	
52	25	42	23	530k	310k	147	81	52	25	42	23	530k	310k	105	45	250	170

*Red coloured values show the number of transitions (in millions) in original object code (MM_i) of a pacemaker without abstractions through SSA tool

*Blue coloured values represent the number of transitions (in millions) in object code (MM_i) of a pacemaker with abstractions through SSA tool

Theorem 3. *The implementation state before and after the merger in abstracted code maps to the same specification state:*

$$\forall w \in S_c :: \left(\mathscr{M}_{erger} \wedge r(w) = S_c\right) \longrightarrow (r(v) = S_c), \quad (29)$$

where $\mathscr{M}_{erger}$ is the set containing mergers in SSA abstracted code and is shown in Table 1 for our case study.

This theorem can be proved for any SSA abstracted code using proof by cases. We have developed proof obligations by considering all possible cases in the implementation of the pacemaker, which are justified by the specification. In actual object code, several repetitive code fragments can be abstracted by SSA and replaced by respective mergers. Consequently, the large state space is reduced, which makes the verification of object code efficient.

The implementation of the pacemaker control program has several numbers of assembly instructions. The specification TS STS_{PM} is nondeterministic and it has different paths. The implementation of a pacemaker is a real-time system that can actually take different paths on run-time. We have considered all such possible paths that can occur during the run-time of object code for this case. Each path taken by the control program or object code has a different length, depending on branches and input behavior. The input-dependent behavior in specification STS_{PM} is implemented with interrupts. We applied SSA on every path of the program to see the difference after abstraction. The reduction in the size of implementation TS is depicted in terms of the number of transitions along each path in Table 3. There are 4 different paths that this specific implementation (object code) can go through with and without the input interrupt. In hardware, there are actually millions of transitions because, at every clock cycle of the controller, the object code is executed and the system state is defined. Along each path, SSA combines the stuttering instructions and reduces the length of object code between two successive states that map to two successive states of the specification. For example, let us consider path 1 in Table 3.

For this path, there are 3 state transitions, with different interrupt sources (s_0, s_0), (s_0, s_1), and (s_1, s_5). We can see in the last row of Table 3 that, in all three state transitions, the numbers of implementation transitions are reduced by almost 50%. A similar effect can be seen in the state transitions of path 2, path 3, and path 4. The table depicts the effect of SSA on all the possible code paths that the pacemaker code can take in real time. The substantial reduction in implementation TS can be observed in Table 3, which ultimately decreases the efforts involved in formal verification of the object code.

Figure 8 shows some statistics obtained while proving the correctness by the theorem prover. It is obvious through the plots that the SSA abstracted object code (mergers) takes

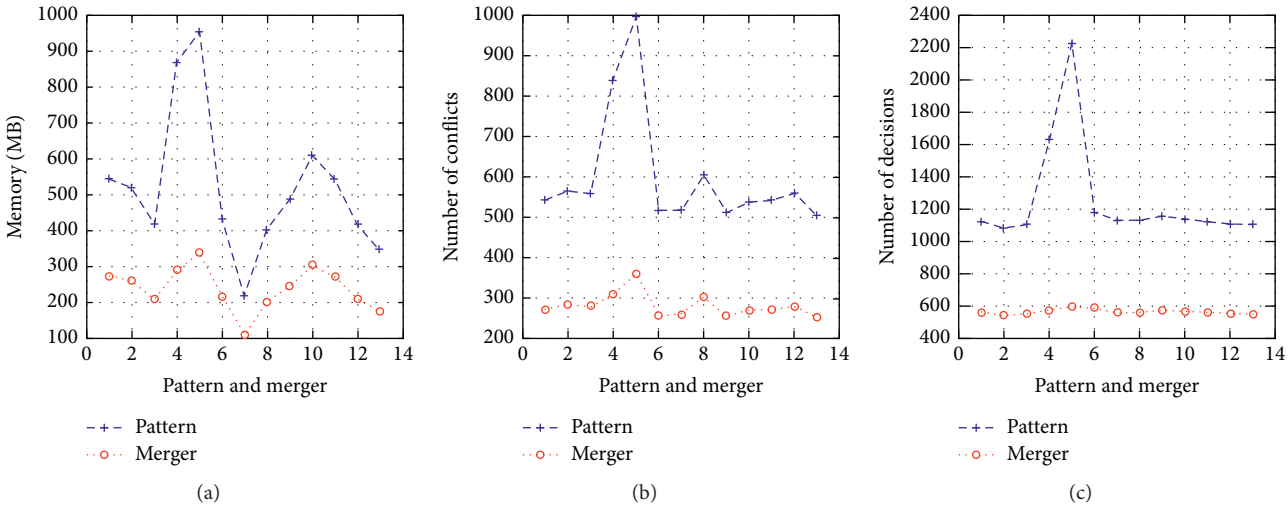

Figure 8: Comparison of some theorem prover statistics for verifying the mergers in abstracted object code versus original object code of a pacemaker in LPC1768.

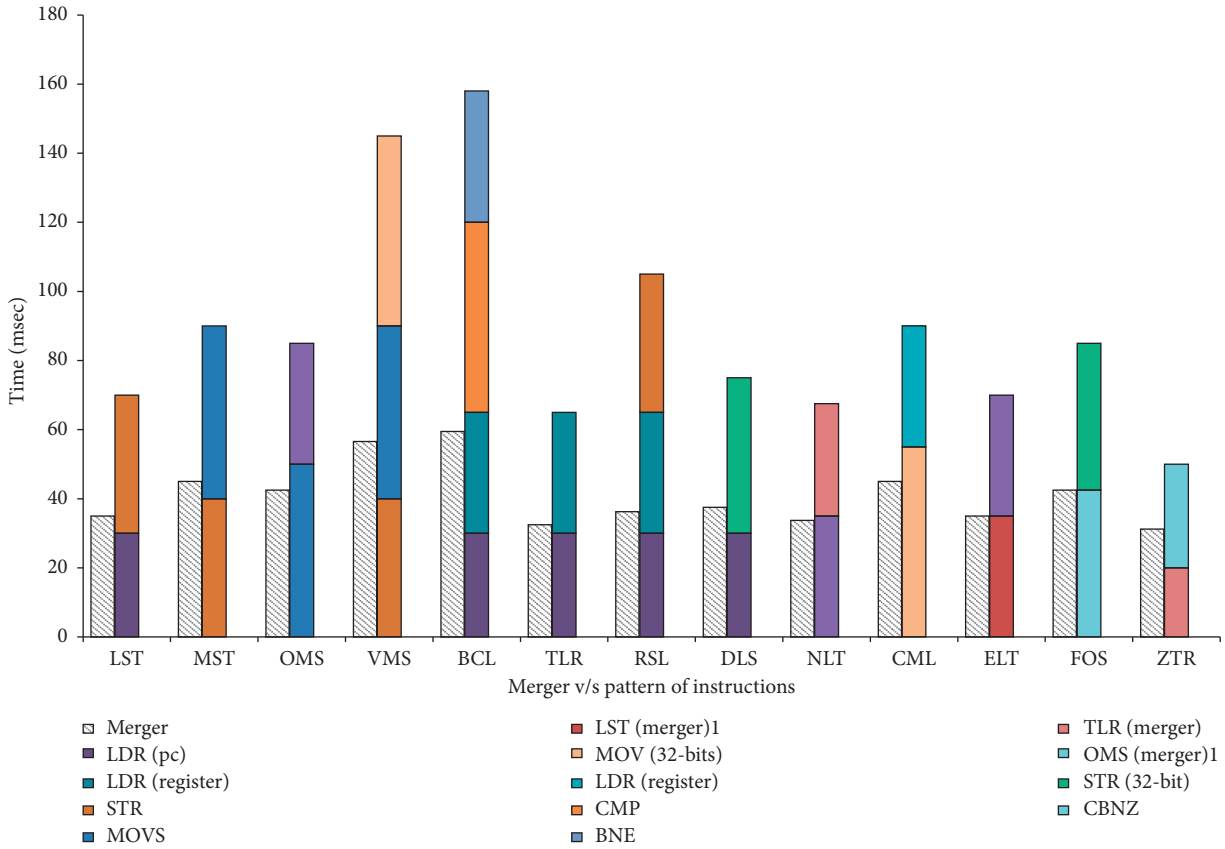

Figure 9: Comparison of time taken by theorem prover for patterns in abstracted object code and original object code of a pacemaker in LPC1768.

lesser memory, number of conflicts, and numbers of decisions as compared to the original object code.

Our results also indicate an obvious improvement in terms of time taken by the proofs. The graph plot in Figure 9 shows the time taken in proving the original code fragments and the SSA merged code. It can be clearly observed that the mergers take lesser time in verifying proofs as compared to the original instructions proofs.

8. Conclusion and Future Work

Verification of object code involves significant overhead in terms of time and efforts, and it is at the same time demanding because of safety concerns. Even in aerospace engineering, only the important safety-critical modules (which are usually the subsets of a whole application) are nominated for object code verification because of the complexity involved. This practice may leave underlying loopholes that can lead to program failure in some cases and eventually lead to unwanted consequences. We propose an approach to abstract the object code statically known as Static Stuttering Abstraction (SSA). It reduces the large size of object code program. The repetitive segments or groups of commands are sorted throughout the program code and combined into an abstract instruction known as merger. The program function is not altered after SSA even during the occurrence of branches or jumps in the original program code. This has been illustrated with the help of Table 3, where SSA is applied on different paths depending on different branching behaviors on run-time. All of these different paths in implementation have been verified to be correct after SSA.

The proposed technique (SSA) contributes to the acceleration of refinement-based verification. Our abstraction technique is targeted for a very large state space like object code. SSA in our case study reduced the code size by 41% in LPC1768 controller and by 23% in Atmega328p controller as shown in Table 2. The difference in the code size reduction is because of two different ISA of two different controllers. The authors in [24] have applied the WFS refinement-based verification without abstraction. Each proof obligation generated by [24] has several associated proof files, depending on the number of object code instructions, which are discharged into Z3. SSA reduces the number of instructions so the numbers of proof files to be generated are reduced significantly. Consequently, the reduced manual efforts help to integrate a detailed and complete coverage of object code verification. In medical domain specifically, object code verification has become inevitable and commercially justified. We verified pacemaker case study and compared our results to [24]. The graph in Figure 9 depicts the comparison of time taken by theorem prover to verify a set of instructions in a pattern in original object code and its merger in updated object code. It is evident that the verification time has been considerably reduced after SSA for each original code fragment. The type of instruction patterns and number of mergers generated for LPC1768-based pacemaker object code are shown in Table 1. The first column of Table 3 shows the repetitive instruction patterns identified by SSA in original code. The second and third columns show the number of lines in the pattern and the number of times the pattern occurs in original code, respectively. The fourth column shows the original instructions opcode. The merger label and merger ASCII opcode are given in the fifth and sixth columns, respectively. The last column shows the binary opcode for merger. We identified fifteen patterns and obtained fifteen mergers for this particular object code implementation. Previously, many abstraction-based techniques have been designed for model checking but none of them targeted object code of a real-time application.

In the future, we intend to combine static and dynamic stuttering abstraction techniques and evaluate the efficacy of the cascaded technique. Dynamic stuttering abstraction is the symbolic simulation of an object code to obtain a reduced state transition system implementation of dynamic stuttering abstraction after the implementation of SSA is expected to solve the problem of an extremely large state space. We also plan to work on automating the WEB refinement process for equivalence checking between specification and implementation, which will eventually aid the refinement process by reducing the time and effort involved in the verification of the object code.

Data Availability

The data used to support the findings of this study are included within the article. In addition, in order to better share the research results, the codes related to the designed SSA tool are available from the corresponding author upon request.

Disclosure

The contents do not necessarily reflect the views of the United States Government.

Conflicts of Interest

The authors declare that there are no conflicts of interest.

Acknowledgments

This work was supported by a grant from the United States Government and the generous support of the American people through the United States Department of State and the United States Agency for International Development (USAID) under the Pakistan-U.S. Science and Technology Cooperation Program.

References

[1] P. Garoche, *Formal Verification of Control System Software*, Vol. 67, Princeton University Press, Berlin, Heidelberg, 2019.

[2] A. O. Gomes and M. Oliveira, "Formal development of a cardiac pacemaker: from specification to code," in *Brazilian Symposium on Formal Methods*, pp. 210–225, Springer, Berlin, Heidelberg, 2010.

[3] S. Brown, "Overview of IEC 61508. Design of electrical/electronic/programmable electronic safety-related systems," *Computing & Control Engineering Journal*, vol. 11, no. 1, pp. 6–12, 2000.

[4] H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman, "Analysis of safety-critical computer failures in medical devices," *IEEE Security & Privacy*, vol. 11, no. 4, pp. 14–26, 2013.

[5] Medical Device Recalls, 2017, https://www.fda.gov/MedicalDevices/Safety/ListofRecalls/ucm535289.htm.

[6] O. Hasan and S. Tahar, "Formal verification methods," in *Encyclopedia of Information Science and Technology*,

pp. 7162–7170, IGI Global, Berlin, Heidelberg, 3rd edition, 2015.

[7] Q. Jabeen, F. Khan, M. N. Hayat, H. Khan, S. R. Jan, and F. Ullah, "A survey: embedded systems supporting by different operating systems," 2016.

[8] N. Shaukat, S. Shuja, S. Srinivasan, S. Jabeen, and M. A. L. Dubasi, "Static stuttering abstraction for object code verification," in *Proceedings of the CYBER 2018: The Third International Conference on Cyber-Technologies and Cyber-Systems*, pp. 102–106, Austin, USA, 2018.

[9] P. Manolios, *Mechanical Verification of Reactive Systems*, University of Texas, Austin, USA, 2001.

[10] U.S. Food and Drug Administration, "Infusion Pump Improvement Initiative," 2010, http://www.fda.gov/downloads/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/UCM206189.pdf.

[11] S. Yovine, "Kronos: a verification tool for real-time systems," *STTT*, vol. 1, no. 1-2, pp. 123–133, 1997.

[12] G. Behrmann, A. David, and K. G. Larsen, W. P. Yi and Y. Wang, Developing UPPAAL over 15 years," *Software: Practice and Experience*, vol. 41, no. 2, pp. 133–142, 2011.

[13] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[14] J. C. Godskesen, K. G. Larsen, and A. Skou, "Automatic verification of real-timed systems using Epsilon," in *Protocol Specification, Testing and Verification XIV*, pp. 323–330, Springer, Boston, MA, 1995.

[15] J. F. Groote and A. Wijs, "An $O (m\log n)$ algorithm for stuttering equivalence and branching bisimulation," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, Heidelberg, pp. 607–624, 2016.

[16] S. Jabeen, S. K. Srinivasan, S. Shuja, and M. A. L. Dubasi, "A formal verification methodology for FPGA-based stepper motor control," *IEEE Embedded Systems Letters*, vol. 7, no. 3, pp. 85–88, 2015.

[17] R. Sumners, "Proof reduction of fair stuttering refinement of asynchronous systems and applications," 2017.

[18] M. Jain and P. Manolios, "An efficient runtime validation framework based on the theory of refinement," 2017.

[19] A. Rabinovich, "Automata over continuous time," *Theoretical Computer Science*, vol. 300, no. 1–3, pp. 331–363, 2003.

[20] J. Klein and C. Baier, "On-the-fly stuttering in the construction of deterministic ω-automata," in *Proceedings of the International Conference on Implementation and Application of Automata*, pp. 51–61, Springer, Berlin, Heidelberg, 2007.

[21] S. Jabeen, S. Srinivasan, and S. Shuja, "Formal verification methodology for real-time field programmable gate array," *IET Computers & Digital Techniques*, vol. 11, no. 5, pp. 197–203, 2017.

[22] P. C. Ölveczky and J. Meseguer, "Abstraction and completeness for real-time maude," *Electronic Notes in Theoretical Computer Science*, vol. 176, no. 4, pp. 5–27, 2007.

[23] M. A. L. Dubasi, S. K. Srinivasan, S. Shuja, and Z. A. Al-Odat, "Refinement checker for embedded object code verification," in *Proceedings of the The Fourth International Conference on Cyber-Technologies and Cyber-Systems*, pp. 81–87, IARIA, Berlin, Heidelberg, 2019.

[24] S. Shuja, S. K. Srinivasan, S. Jabeen, and D. Nawarathna, "A formal verification methodology for DDD mode pacemaker control programs," *Journal of Electrical and Computer Engineering*, vol. 57, 2015.

[25] S. Nejati, A. Gurfinkel, and M. Chechik, "Stuttering abstraction for model checking," in *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pp. 311–320, IEEE, Berlin, Heidelberg, 2005.

[26] D. D. Dunlop and V. R. Basili, "A comparative analysis of functional correctness," *ACM Computing Surveys*, vol. 14, no. 2, pp. 229–244, 1982.

[27] Z. Manna and A. Pnueli, "Axiomatic approach to total correctness of programs," *Acta Informatica*, vol. 3, no. 3, pp. 243–263, 1974.

[28] C. A. R. Hoare, "ViewpointRetrospective," *Communications of the ACM*, vol. 52, no. 10, pp. 30–32, 2009.

[29] D. Gries, *The Science of Programming*, Springer Science & Business Media, Berlin, Heidelberg, 2012.

[30] A. Chlipala, "The bedrock structured programming system," in *ACM Sigplan Notices*, vol. 48, no. 9, pp. 391–402, ACM, 2013.

[31] O. Mraihi, W. Ghardallou, A. Louhichi, K. Bsaies, and M. Ali, "Computing preconditions and postconditions of while loops," in *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, Springer, Berlin, Heidelberg, pp. 173–193, 2011.

[32] S. S. Epp, *Discrete Mathematics with Applications*, Cengage Learning, Berlin, Heidelberg, 2010.

[33] N. Robbins, *Beginning Number Theory*, Jones & Bartlett Learning, Berlin, Heidelberg, 2006.

[34] Heart Pacemaker, 2020, https://medlineplus.gov/ency/article/007369.htm.