Refinement Checker for Embedded Object Code Verification

Mohana Asha Latha Dubasi*, Sudarshan K. Srinivasan†, Sana Shuja‡, Zeyad A. Al-Odat†
*Configurable IP and Chassis Group, Intel Corporation, Hillsboro, OR, USA
†Electrical and Computer Engineering, North Dakota State University, Fargo, ND, USA
‡Department of Electrical Engineering, COMSATS University, Islamabad, Pakistan
Emails: *dubasi.asha@gmail.com, †sudarshan.srinivasan@ndsu.edu, ‡sanashuja@comsats.edu.pk, †zeyad.alodat@ndsu.edu

***Abstract*—We present a formal verification methodology that automates the process to check for correctness of low-level real-time interrupt-driven object code programs. Automation helps in the verification of large-scale programs. Our methodology is based on the theory of Well-Founded Simulation (WFS) refinement, where both the formal specification and implementation are modeled as transition systems (TSs). WFS refinement is used as the notion of correctness and defines what it means for an implementation TS to satisfy its specification TS. WFS refinement has key features like stuttering and refinement maps. Stuttering aids in the abstraction of the state space of the implementation TS. Refinement map bridges the abstraction gap between the two systems. The efficiency and scalability of the approach is demonstrated on several device object code case studies.**

Keywords–embedded devices; formal verification; refinement-based verification; verification of object code, WFS refinement.

I. INTRODUCTION

Correctness of software used in safety-critical systems continues to be a critical challenge. For example, between the years 2005-2019, the U.S. Food and Drug Administration (FDA) [1] issued 52 medical device Class-1 recalls due to software issues. Class-1 recalls are applied when the use of the device is determined to cause serious adverse health consequences or death. It is now well-established that formal verification methods are a requirement to ensure software safety.

Our domain of interest is the verification of embedded software, which is largely what is used in control of medical devices, surgical robots, avionics equipment, etc. While many formal verification methods exist for reasoning about higher-level [2]–[4] software models and source code, there is currently a gap in the applicability of formal verification techniques that can efficiently scale and handle the low-level complexity of object code, which is the low-level code that is directly executed by the micro-controller embedded in the device used to perform control and other functions. It is definitely insufficient to apply formal verification only to source code as there are many sources of errors in the process of generating object code from source code that can compromise the safety of object code. The problem domain addressed in this work is the verification of embedded object code programs.

Typically, the objective of a formal verification methodology is to verify an implementation (the artifact to be verified, here, object code) against a specification (the artifact that captures the requirements to be satisfied by the implementation). Previously, we have developed a formal verification methodology for object code verification. The methodology is based on the theory of Well Founded Simulation (WFS) refinement [5]. In the context of WFS refinement, both the implementation and specification are modeled as Transition Systems (TSs: a mathematical modeling framework for code that is based on states of the program and transitions between states). WFS refinement essentially defines what it means for an implementation TS to correctly implement a specification TS. It has been explained in [5] how the low-level code is represented as an implementation TS and how it is the implementation of the high-level TS that acts as the specification.

The methodology was demonstrated by manually generating the required proof obligations for checking WFS refinement. However, this is insufficient for large programs. In this paper, we address this gap by proposing an algorithm for automatic WFS refinement checking optimized for object code verification. This algorithm checks for safety based on WFS refinement. Safety informally means that if the implementation makes progress, the result of that progress satisfies the specification requirements. The algorithm has been implemented and the automated tool flow has been applied to several object code control programs to demonstrate the effectiveness of the approach.

The rest of this paper is organized as follows. Related work is given in Section II. An overview of WFS refinement and related concepts is presented in Section III. The algorithm for safety verification is presented in Section IV. Results and conclusions are given in Sections V and VI, respectively.

II. RELATED WORK

A large gap exists between high-level system models and the low-level actual code (*i.e.*, object-code), which is executed on the embedded device. The number of states and transitions in the high-level system models is typically less than 100, whereas, in the low-level models the number of states and transitions may be in the order of millions. Catching design bugs early in the design cycle of the system-level models is very useful. To bridge the gap between system-level models and actual code, model-driven approaches are adopted. Here, the high-level system models are represented as source code which is developed using platform-independent synthesis tools. The source code is then augmented with the device peripheral information and then compiled and assembled to generate the object code. During this process, numerous errors can creep into the object code compromising its safety. Our work is targeted at bridging the gap between the real-time high-level models and real-time object code and also ensuring that the object-code is safe.

There has been a lot of previous work in developing theory and optimized techniques for refinement based verification. A notion of refinement based on stuttering trace containment to verify the concurrent programs have been developed in [6]. A refinement-based testing method have been developed in [7], which checks for the functional correctness of hardware and

 ISBN: 978-1-61208-743-6

low-level software. A characterization of stuttering bisimulation has been proposed in [8], here, stuttering bisimulation is a notion of correctness that defines what it means for two transition systems to be equivalent. Stuttering is accounted for in this work. An algorithm that checks equivalence between two transition systems which accounts for stuttering has been presented in [9]. Derrick *et al*. [10] have presented refinement checker for Z, which is a language that is used to express computer programs. These computer programs are system-level models. Gibson-Robinson *et al*. [11] have presented a refinement checker that check if one system is a refinement of the other, where both the systems are expressed as transition systems. This checker is used on software models like concurrent systems. However, these do not consider real-time interrupt-driven object code. In this work, we use WFS refinement which has a very nice property. The check is local, i.e., it is sufficient to reason about a single step of the implementation and specification. Since object code typically contains millions of transitions, this property can be exploited by reasoning about one transition at a time. This inturn reduces the verification burden. WFS refinement has two features: refinement-map and stuttering. Stuttering helps in abstraction techniques on the TS which reduces the state and path from exploding. Refinement-map bridges the gap between the system-level model and low-level object-code.

Etessami [12] and Dax *et al*. [13] have proposed specification languages for expressing stuttering-invariant properties, which are properties that do not distinguish behaviors of systems that differ only due to stuttering. The properties are verifiable using a model checker. Our work is complimentary to their approach, in that our goal is to exploit stuttering through abstractions to make verification more efficient and scalable.

Shaukat *et al*. [14] have presented an abstraction technique that helps to reduce the object code instructions statically. A number of tools exists [3] [4] [15] to verify real-time high-level models. UPPAAL-based tools like [16] [17] also exist. Al-Qtiemat *et al*. [18] have presented a methodology to generate the formal specification models from natural language requirements. The specification models, useful for refinement verification, are expressed as transition systems. These refinement approaches for real-time systems are targeted at high-level models and do not consider the use of refinement-map and stuttering. Since we incorporate refinement-map and stuttering, our approach is unique and applicable to the verification of low-level implementation such as object-code.

Jabeen *et al*. [19] have used the theory of refinement for the verification of FPGA-based stepper motor control using proof obligations. Manually developing proof obligations for real-time interrupt driven object code is time consuming because of the size of the instructions and may introduce human errors. In contrast, we address the automation of refinement-based verification.

The main goal of our work is verification of real-time object code against it's real-time high-level model. The real-time object code does not handle floating point numbers and C code. The goal is achieved by employing symbolic simulation on object code and this is a standard.

III. BACKGROUND

WFS refinement is a notion of correctness which describes how an implementation system is correct with respect to its specification system. The specification is a mathematical model that describes the behavior of the system in high-level. Usually, the systems that are to be verified are represented as transition systems (TSs).

Definition 1. *[20] A transition system (TS) M is a 3-tuple $\langle S, R, L\rangle$, where S is the set of states, L is a labeling function that defines what is visible at each state and R is the transition relation that defines the state transitions. T is left-total.*

The formulation of the correctness properties and the completeness of the properties with respect to the input language is shown in [20].

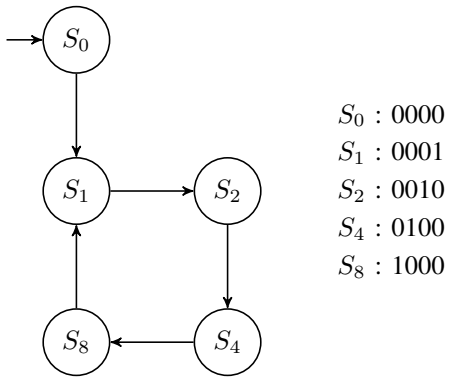

Figure 1. Stepper motor control specification TS

Stepper motor control is used as an example to describe object code verification using WFS refinement. Stepper motors are used in safety-critical applications which include medical devices like infusion pump, robotics like surgical robots, process control, etc. A stepper motor is a brushless DC electric motor, which contains 4 or 6 leads. Discrete rotation of the motor shaft is the result of current pulses that are applied to the motor. A repeating sequence of values such as 0001, 0010, 0100, 1000, 0001, etc, to the leads, cause the motor to spin. Software, that generates the above sequence, can be executed on the microcontroller which is interfaced to the motor. Figure 1 shows the specification TS (M_S) for 4-lead stepper motor control. The states are represented as S_0, S_1, S_2, S_4, S_8. The transition relation determines the direction of the shaft. The labeling function gives the values of the leads, which determine the state.

The implementation model for the stepper motor control would be the object code. This is obtained by generating a function for each instruction that describes the effect of the instruction on the state of the microcontroller. The state of the microcontroller is not as simple as the specification states (as shown in Figure 1), but consists of registers, flags, and memory of the microcontroller. The set of all such functions along with the initial state of the microcontroller defines the TS model of the implementation. The implementation model consists of millions of transitions because of the various possible values that the registers, flags, memory and special registers of the microcontroller can have during the execution of the object

 ISBN: 978-1-61208-743-6

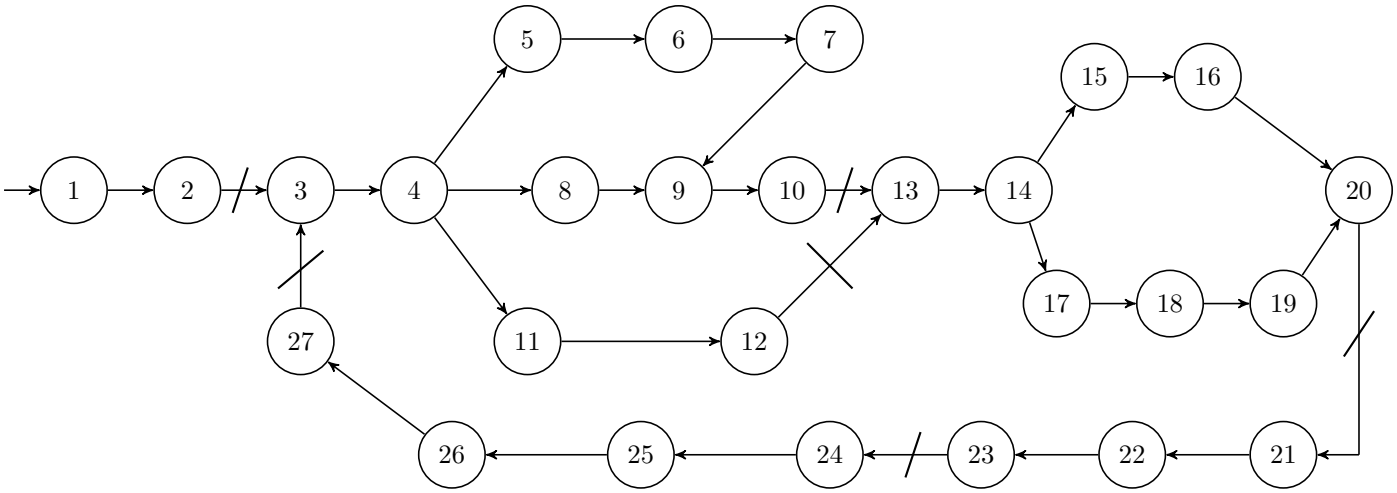

Figure 2. Stepper motor control implementation TS

code program. A details description about the implementation level can be found in [5]. An example of the implementation of the stepper motor control is shown in Figure 2 for the specification in Figure 1.

A detailed description of the WFS refinement is provided in [20]. Below is the definition of WFS

Definition 2. *[20]* $B \subseteq S \times S$ *is a WFS on TS* $\mathcal{M} = \langle S, T, L \rangle$ *iff:*

(1) $\langle \forall s, w \in S :: sBw \quad :: \quad L(s) = L(w) \rangle$; and

(2) There exists functions, $rankl : S \times S \times S \to \mathbb{N}$, $rankt : S \times S \to W$, such that $\langle W, \lessdot \rangle$ is well-founded, and $\langle \forall s, u, w \in S :: sBw \wedge sTu \quad ::$

(a) $\langle \exists v :: wTv \wedge uBv \rangle \vee$

(b) $(uBw \wedge rankt(u, w) \lessdot rankt(s, w)) \vee$

(c) $\langle \exists v :: wTv :: sBv \wedge rankl(v, s, u) < rankl(w, s, u) \rangle \rangle$

Here, condition 1 states that for a WFS relation between s and w, they have to have the same labels. In condition 2, case (a) denotes the non-stuttering transition on the implementation side, case (b) denotes the stuttering on the specification side and case (c) denotes stuttering on the implementation side. Progress on the model is denoted by a non-stuttering transition where the states have different labels. Whereas in the case of stuttering transition the states have the same label. $rankt$ and $rankl$ are rank functions (discussed later).

When the specification and implementation systems are modeled as TSs, a step in the specification could correspond to multiple steps in the implementation. Hence, *stuttering is a phenomenon where multiple but finite transitions of implementation can match to the same specification transition*. WFS refinement has two key features: refinement-map and stuttering. Refinement-map, r, is a function that, given an implementation state, gives the corresponding specification state. A function like refinement-map is needed to bridge the abstraction gap between the implementation and specification systems.

The advantage of using WFS refinement is that it is sufficient to reason about single steps of the implementation and specification to check for correctness and find bugs. This makes WFS refinement applicable to deal with the complexity of object code. To avoid deadlock situations, in stuttering transitions, a witness function called rank is designed such that it decreases with each transition. In definition 2, two rank functions $rankt$ and $rankl$ corresponds to stuttering on specification and implementation side respectively.

Next is the definition of WFS refinement.

Definition 3. *[20] (WFS Refinement) Let* $M = \langle S, T, L \rangle$, $M' = \langle S', T', L' \rangle$, *and* $r : S \to S'$. *We say that* M *is a WFS refinement of* M' *with respect to refinement-map* r, *written* $M \sqsubseteq_r M'$, *if there exists a relation,* B, *such that* $\langle \forall s \in S :: sBr(s) \rangle$ *and* B *is a WFS on the TS* $\langle S \uplus S', T \uplus T', \mathcal{L} \rangle$, *where* $\mathcal{L}(s) = L'(s)$ *for* s *an* S' *state and* $\mathcal{L}(s) = L'(r(s))$ *otherwise.*

In the above definition, M and M' are the implementation and the specification TS respectively and r is the refinement-map function. If stuttering on the specification side is not considered, the definition can be interpreted as every implementation transition should either be a stuttering or a non-stuttering transition. If it can be proved that an implementation TS is a WFS refinement of a specification TS, then every behavior of the implementation is guaranteed to match a behavior of the specification. If the implementation TS has a label that does not match to its specification TS or a transition that does not match in the specification TS, then these situations correspond to a bug in the implementation model (here, $M \not\sqsubseteq_r M'$). If M is a WFS refinement of M' then, a state in M cannot be related to more than one state in M' (as the refinement-map is a function which is used to relate states of M to M').

From Figures 1 and 2, it is clear that the specification and the implementation do not have the same transition behavior. The refinement-map function for the stepper motor control program projects the 4-bits in a register in the microcontroller whose values controls the pins that are in turn connected to the leads of the stepper motor. The implementation TS (M_I) takes several steps (or transitions) to match a single transition of the specification TS. Here, once a refinement-map is constructed, in WFS refinement verification the idea is to look at each

transition. For example, consider an implementation transition $\langle w, v\rangle$ where w, v belong to the set of implementation states. The transition should capture one of the following options. One option is that the implementation transition should match to the same specification state, i.e., $r(w) = r(v) = s$ where $r()$ is the refinement-map and s is a specification state. This option is called a stuttering implementation transition. The second option is that the implementation transition should match a specification transition, i.e., $r(w) = s$ and $r(v) = u$ where $\langle s, u\rangle$ is a transition in specification. This option is called a non-stuttering implementation transition. Using the refinement-map function on the implementation TS (Figure 2) gives that states 1 - 2 translate to state S_0 of the specification TS (Figure 1), states 3 -12 translate to S_1, states 13 - 20 to S_2, states 21 - 23 to S_4 and states 24 - 27 to S_8. The non-stuttering transition in implementation TS (M_I) (Figure 2) are shown with dashed arrows. For object code verification, stuttering rarely occurs on the specification side as the implementation typically has millions of transitions when compared to specification. Hence, case (b) of definition 2 is ignored.

In Figure 2, it can be noticed that many paths in the implementation lead to a specific non-stuttering step. All these finite paths are termed as stuttering segments. *Stuttering segment* π of $\langle w, v\rangle$ [5] where $\langle w, v\rangle$ is a non-stuttering transition can be described as a sequence of transitions in which $\langle w, v\rangle$ is preceded by zero to many stuttering transition(s) and another non-stuttering transition. The least length of a stuttering segment is one. This occurs when a non-stuttering step is preceded by another non-stuttering step. The stuttering segment then only consists of one transition, which is the non-stuttering step. Also, a non-stuttering step can have many stuttering segments. For the TS shown in Figure 2, the stuttering segments of $\langle 10, 13\rangle$ are:

1) $\{\langle 2,3\rangle, \langle 3,4\rangle, \langle 4,5\rangle, \langle 5,6\rangle, \langle 6,7\rangle, \langle 7,9\rangle, \langle 9,10\rangle, \langle 10,13\rangle\}$
2) $\{\langle 2,3\rangle, \langle 3,4\rangle, \langle 4,8\rangle, \langle 8,9\rangle, \langle 9,10\rangle, \langle 10,13\rangle\}$

Object code contains millions of transitions, hence, applying suitable abstraction techniques on the stuttering segments helps to deal with state explosion problem. This reduces the verification problem to analysis of stuttering segments.

IV. Automated WFS Refinement for Object-Code

This section presents a procedure for automating WFS refinement for object code verification. According to definition 2, a TS satisfies WFS when either one of two conditions are satisfied by all transitions. Usually, in real-time object code verification, stuttering does not occur on the specification system. Hence, the refinement-based correctness formula can be reduced to,

$$\begin{aligned}&\langle \forall w \in \text{object-code} :: v = \text{object-code-step}(w) \wedge s = r(w)\\ &\wedge u = \text{SPEC-step(s)} \wedge \langle s, u\rangle \in \text{SPEC then}\\ &\text{(i)}\ \ r(v) = s \text{ (for stuttering transition) or}\\ &\text{(ii)}\ \ r(v) = u \text{ (for a non-stuttering transition) } \rangle\end{aligned} \tag{1}$$

Here, once a refinement-map is constructed, in WFS refinement verification the idea is to look at each transition. Let $\langle w, v\rangle$ be an implementation transition where w, v belong to the set of implementation states. To satisfy the refinement-based correctness formula, the transition should capture one of the options of being either a stuttering transition or a non-stuttering transition. If the implementation transition shows a behavior that is neither stuttering nor non-stuttering then it indicates the presence of a bug in the implementation TS.

Typically object code (implementation TS (M_I)) consists of millions of transitions where a large portion of these transitions are of stuttering in nature. Hence, abstraction based on these stuttering transitions may be applied on the implementation TS (M_I). Applying stuttering abstractions on the TS makes the verification process faster and much more efficient.

```
1:  procedure CheckWFSRef(R_I, R_S, S_S, r)
2:      R_U ← NULL;
3:      for i ← 1 to |R_I| do
4:          ⟨w, v⟩ ← R_I[i];
5:          s ← r(w);
6:          match ← FALSE;
7:          if s ∈ S_S then
8:              if r(v) = r(w) then
9:                  match ← TRUE;
10:             else
11:                 for j ← 1 to |R_S| do
12:                     ⟨s, u⟩ ← R_S[j];
13:                     if r(v) = u then
14:                         match ← TRUE;
15:                         break;
16:         if match = FALSE then
17:             R_U ← R_U ∪ ⟨w, v⟩;
18:     return R_U;
```

Figure 3. Procedure for Checking WFS Refinement

The algorithm in Figure 3 presents a procedure that performs WFS refinement checking on the abstracted object code TS, which is the implementation TS. The inputs to the procedure include a list of transitions of R_I (implementation TS), a list of transitions of R_S (specification TS), a set of states of the specification (S_S) and the refinement-map r. R_U is the counterexample set, which the procedure will populate with implementation transitions that do not satisfy the WFS refinement correctness criteria (1). R_U is initially empty. The procedure iterates through each transition in R_I. The transitions of the implementation are of the form $\langle w, v\rangle$. Variable 's' is assigned to be the value refinement-map (w) (line 5). Predicate *match* is used to keep track of whether the implementation transition has found a matching specification transition, or is determined to be stuttering, or is neither.

A check is performed on 's' to see if it belongs to the set of states of specification (S_S). If 's' does not belong to S_S, then the w state has no corresponding specification state and therefore points to an error and the transition is added to R_U. When 's' exists in S_S, a check has to be performed on the implementation transition to see if it is a stuttering transition or a non-stuttering transition. In case the transition is a stuttering transition, the predicate *match* is set to true and the procedure proceeds to the next transition in the implementation.

If the transition is a non-stuttering transition, the procedure iterates through the specification transitions $\langle s, u\rangle$. If a u is found such that $r(v)$ is equal to u, then *match* is assigned true. Once a match is found, the procedure exits iterating through the specification transitions and moves on to the next implementation transition. If for a non-stuttering transition, $r(v)$ does not match with any u, then this points to an error in the implementation and the corresponding implementation transition is appended to R_U. When all the transitions of the implementation have been checked, the procedure ends by returning the list R_U (line 18).

The time complexity of this algorithm is $\mathcal{O}(|R_I||R_S|)$. The outer for loop has length of R_I passes. The if condition on line 7 has length of S_S passes. The inner for loop has length of R_S passes. The complexity of the algorithm is $|R_I|*(|S_S|+|R_S|)$. Usually, the number of transitions of the specification ($|R_S|$) is greater than equal to the number of states of specification ($|S_S|$) depending on the application. Hence, the overall time complexity is $\mathcal{O}(|R_I||R_S|)$.

V. RESULTS

Case Study - 1: The effectiveness of the algorithm presented in this paper were demonstrated on 22 different object code programs for stepper motor control. In this paper, three sequences of stepper motor control that uses 4 leads are used to develop the benchmarks. Double stepping sequence an be described as $\langle 0011\rangle$, $\langle 0110\rangle$, $\langle 1100\rangle$, $\langle 1001\rangle$, $\langle 0011\rangle$, so on. Full stepping sequence an be described as $\langle 0001\rangle$, $\langle 0010\rangle$, $\langle 0100\rangle$, $\langle 1000\rangle$, $\langle 0001\rangle$, so on. Half stepping sequence an be described as $\langle 0001\rangle$, $\langle 0011\rangle$, $\langle 0010\rangle$, $\langle 0110\rangle$, $\langle 0100\rangle$, $\langle 1100\rangle$, $\langle 1000\rangle$, $\langle 1001\rangle$, $\langle 0001\rangle$, so on.

The programs were developed to run on an ARM Cortex-M3 based NXP LPC1768 [21] microcontroller. PORT 2 of the LPC1768 was used to connect the leads of the stepper motor using an electronic circuit. Repetitive Interrupt Timer (RIT) was used to generate interrupts at regular intervals of time in some of the benchmarks to implement the timing requirements for stepper motor control.

Table I shows the verification statistics for the benchmarks. The benchmark name indicates the type of control used. "Full", "Double", and "Half" indicate full stepping, double stepping, and half stepping were used, respectively. "RIT" indicates that the interrupts were generated by Repetitive Interrupt Timer (RIT) to implement the timing delays for the motor control. "noRIT" indicates that instead of the RIT timer, code was to implement timing delays. "clock" and "anti" indicate that the motor was controlled clockwise and anti-clockwise, respectively. The table gives statistics for both correct and buggy versions of the controllers. "FuncBug" indicate that the object code error was a functional error. Column 3 gives the number of transitions of the object code TS. Column 4 gives the number of transitions in the abstract TS (which is generated by applying suitable abstraction techniques). Column 5 gives the time taken to perform WFS refinement.

Case Study - 2: The effectiveness of the algorithm presented in this paper were also demonstrated on industrial example, an infusion pump. An infusion pump is a medical device that can give controlled dosage of medications, like opioids, insulin, etc, or nutrients into the patients's circulatory system intravenously.

TABLE I. VERIFICATION STATISTICS

S.No	Object Code Benchmarks	# of Trans. of MM_I [million]	# of Trans. of of Abstract MM^a	WFS Verifi. Time [millisec]
1	Full-RIT-clock	2.5	10	3
2	Full-RIT-anti	2.5	10	3
3	Double-RIT-clock	2.5	10	4
4	Double-RIT-anti	2.5	10	3
5	Half-RIT-clock	4.5	18	4
6	Half-RIT-anti	4.5	18	3
7	Full-noRIT-clock	82.5	10	3
8	Full-noRIT-anti	82.5	10	4
9	Double-noRIT-clock	82.5	10	4
10	Double-noRIT-anti	82.5	10	6
11	Half-noRIT-clock	148.5	18	3
12	Half-noRIT-anti	148.5	18	5
13	FuncBug-Full -RIT-clock	2.5	10	5§
14	FuncBug-Full -RIT-anti	2.5	10	3§
15	FuncBug-Double -RIT-clock	2.5	10	4§
16	FuncBug-Double -RIT-anti	2.5	10	4§
17	FuncBug-Half -RIT-clock	4.5	18	5§
18	FuncBug-Half -RIT-anti	4.5	18	3§
19	FuncBug-Full -noRIT-clock	99	20	3§
20	FuncBug-Full -noRIT-anti	99	20	3§
21	FuncBug-Double -noRIT-clock	82.5	10	4§
22	FuncBug-Double -noRIT-anti	82.5	10	3§

§ indicates the time taken to generate counterexample

The program was developed for Alaris Medley 8100 LVP module infusion pump [22], [23] for our experiments. Pulse width modulation technique is used by this pump to control the dosage delivered. The pulse width modulation control code was implemented for the Alaris pump on an ARM Cortex M3 based LPC 1768 micro-controller. The pump was interfaced to the micro-controller so that our code can control the pump. The formal specifications for the pump control software was developed based on the requirements in [24].

TABLE II. VERIFICATION STATISTICS FOR INFUSION PUMP CONTROLLER

S.No	Object Code Benchmarks	# of Trans. of MM_I [million]	WFS Verifi. Time [millisec] of Abstract MM^a
1	IPC	24.3	3
2	IPC-FuncBug1	20.25	2§
3	IPC-FuncBug2	24.3	7§
4	IPC-FuncBug3	27	5§

§ indicates the time taken to generate counterexample

Table II shows the verification statistics for the infusion pump control case study. The transition system of the pump's control code had about 24.3 million transitions. The table gives statistics for both correct and buggy versions of the controller. "FuncBug" indicate that the object code error was a functional error. Column 3 gives the number of transitions of the object code TS. Column 4 gives the time taken to perform WFS

refinement.

The code is not small snippets, but a working infusion pump setup where the code is used to run the infusion pump. It has been demonstrated that our method works with control code. In general, code can be arbitrarily large, we have not explored the scalability of our approach to problems in the order of 100K lines of code.

For case studies 1 and 2, the verification experiments were performed on an Intel Core i7 3.1 GHz processor with 8GB memory. Using suitable abstraction on the stuttering segments, the number of transitions in the implementation TS have been reduced from hundred of millions to less than 50 transitions. Because of this huge reduction in the size of the state space, it took less than a second to perform WFS refinement checking on the abstracted TS.

VI. Conclusion

The effectiveness of the refinement checker has been demonstrated from the verification results. Applying suitable abstraction on the stuttering segments reduces the number of transitions in the implementation TS for all the benchmarks. The reduced size of the implementation TS enables detecting and correcting the errors very easily since the verification tools are often used multiple times in practice.

This paper presents only the safety verification technique. It is intended to extend this work further by including techniques to detect deadlock errors. If the code is not making progress with respect to the specification, then such a behavior is known as deadlock.

Acknowledgment

This publication was funded 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 & Technology Cooperation Program. The contents do not necessarily reflect the views of the United States Government.

References

[1] US Food and Drug Administration (FDA), "Medical device recalls," https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm, 2019, last accessed: April 2019.

[2] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," STTT, vol. 1, no. 1-2, 1997, pp. 134–152.

[3] M. Bozga and et al., "Kronos: A model-checking tool for real-time systems (tool-presentation for ftrtft '98)," in Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 14-18, 1998, Proceedings, ser. Lecture Notes in Computer Science, A. P. Ravn and H. Rischel, Eds., vol. 1486. Springer, 1998, pp. 298–302.

[4] J. C. Godskesen, K. G. Larsen, and A. Skou, "Automatic verification of real-time systems using epsilon," in Protocol Specification, Testing and Verification XIV, Proceedings of the Fourteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Vancouver, BC, Canada, 1994, ser. IFIP Conference Proceedings, S. T. Vuong and S. T. Chanson, Eds., vol. 1. Chapman & Hall, 1994, pp. 323–330.

[5] M. A. L. Dubasi, S. K. Srinivasan, and V. Wijayasekara, "Timed refinement for verification of real-time object code programs," in Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Kroening, Eds., vol. 8471. Springer, 2014, pp. 252–269.

[6] S. Ray and R. Sumners, "Specification and verification of concurrent programs through refinements," J. Autom. Reasoning, vol. 51, no. 3, 2013, pp. 241–280.

[7] M. Jain and P. Manolios, "An efficient runtime validation framework based on the theory of refinement," CoRR, vol. abs/1703.05317, 2017.

[8] K. S. Namjoshi, "A simple characterization of stuttering bisimulation," in Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings, 1997, pp. 284–296.

[9] J. F. Groote and A. Wijs, "An o(m\log n) algorithm for stuttering equivalence and branching bisimulation," in Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, ser. Lecture Notes in Computer Science, M. Chechik and J. Raskin, Eds., vol. 9636. Springer, 2016, pp. 607–624.

[10] J. Derrick, S. North, and A. J. H. Simons, "Building a refinement checker for Z," ser. EPTCS, J. Derrick, E. A. Boiten, and S. Reeves, Eds., vol. 55, 2011, pp. 37–52.

[11] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3: a parallel refinement checker for CSP," STTT, vol. 18, no. 2, 2016, pp. 149–167.

[12] K. Etessami, "Stutter-invariant languages, omega-automata, and temporal logic," in Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings, 1999, pp. 236–248.

[13] C. Dax, F. Klaedtke, and S. Leue, "Specification languages for stutter-invariant regular properties," in Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings, ser. Lecture Notes in Computer Science, Z. Liu and A. P. Ravn, Eds., vol. 5799. Springer, 2009, pp. 244–254.

[14] N. Shaukat, S. Shuja, S. K. Srinivasan, S. Jabeen, and M. A. L. Dubasi, "Static stuttering abstraction for object code verification," in CYBER 2018, The Third International Conference on Cyber-Technologies and Cyber-Systems, Athens, Greece. IARIA, Nov 2018, pp. 102–106.

[15] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Developing UPPAAL over 15 years," Softw., Pract. Exper., vol. 41, no. 2, 2011, pp. 133–142.

[16] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed I/O automata: a complete specification theory for real-time systems," in Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010, K. H. Johansson and W. Yi, Eds. ACM, 2010, pp. 91–100.

[17] A. Boudjadar, J. Bodeveix, and M. Filali, "Compositional refinement for real-time systems with priorities," in 19th International Symposium on Temporal Representation and Reasoning, TIME 2012, Leicester, United Kingdom, September 12-14, 2012, B. C. Moszkowski, M. Reynolds, and P. Terenziani, Eds. IEEE Computer Society, 2012, pp. 57–64.

[18] E. Al-Qtiemat, S. K. Srinivasan, M. A. L. Dubasi, and S. Shuja, "A methodology for synthesizing formal specification models from requirements for refinement-based object code verification," in CYBER 2018, The Third International Conference on Cyber-Technologies and Cyber-Systems, Athens, Greece. IARIA, Nov 2018, pp. 94–101.

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

[20] P. Manolios, "Mechanical verification of reactive systems," Ph.D. dissertation, University of Texas at Austin, 2001.

[21] "Keil cortex-m evaluation board comparison," http://www.keil.com/arm/boards/cortexm.asp, last accessed: April 2019.

 ISBN: 978-1-61208-743-6

[22] CareFusion, Alaris PC Unit, Alaris Pump Module, Technical Service Manual, CareFusion, 2010.

[23] ALARIS Medical Systems, Inc., Directions for use. Pump Module, 8100 series, ALARIS Medical Systems, Inc., 2004.

[24] Y. Zhang, R. Jetley, P. L. Jones, and *et al*., "Generic safety requirements for developing safe insulin pump software," Journal of Diabetes Science and Technology, vol. 5, no. 6, 11 2011, pp. 1403–1419.