Static Stuttering Abstraction for Object Code Verification

Naureen Shaukat*, Sana Shuja*, Sudarshan Srinivasan†, Shaista Jabeen* and Mohana Asha Latha Dubasi†
*Department of Electrical Engineering
COMSATS University, Islamabad , Pakistan
Email: nsawan23@gmail.com, sanashuja@comsats.edu.pk, shaista.sj@comsats.edu.pk
†Department of Electrical and Computer Engineering
North Dakota State University, Fargo, USA
Email: sudarshan.srinivasan@ndsu.edu, mohanaashalatha.duba@ndsu.edu

Abstract—**The biggest challenge in the formal verification of an embedded system software is the complexity and large size of the implementation. The problem gets even bigger when the embedded system is an Internet of Things (IoT) device that is running intricate algorithms. In Refinement-based verification, both specification and implementation are expressed as transition systems. Each behavior of the implementation transition system is matched with the specification transition system with the help of a refinement map. The refinement map can only project those values that are responsible to label the current state of the system. When the refinement map is applied at object code level, several instructions map to a single state in the specification transition system called stuttering instructions. The concept of Static Stuttering Abstraction (SSA) is a novel idea that focuses on filtering common multiple segments of these stuttering instructions. The patterns are then replaced by mergers that preserve the behavior of the original object code and extensively reduce the size of the object code. The smaller code size also gives the lesser number of stuttering transitions and eventually more discernible matching between the specification and implementation of transition systems. We have implemented SSA technique on two platforms using infusion pump as a case study and the technique has proved consistent in considerably reducing the size and complexity of the implementation transition system.**

Keywords–Formal verification; static stuttering abstraction; stuttering instructions; refinement map; infusion pump.

I. INTRODUCTION

One of the pivotal entities in the ecosystem of Internet of Things (IoT) is an embedded system due to its capability of receiving data from sensors and making decisions independently. From a cell phone to an automobile, all are comprised of an embedded system; that collects, and senses data received, collates that data to be analyzed, and consequently perform necessary functions. The functionality of the embedded system not only encompasses basic I/O, but it also involves complex communication protocols that consent other indispensable peripherals to communicate over shared buses and gateways. Hence, with IoT embedded devices are built on intricate algorithms, which make the verification process even more complicated. Safeguarding these embedded systems against errors is an inevitable task especially in those devices that prevent life-threatening ailments like pacemakers and insulin pumps. Such devices can cause severe consequences if the software or hardware malfunctions, making these medical devices safety critical. For example, from 2001 to 2017, the Food and Drug Administration (FDA) has issued 54 Class-1 recalls on infusion pumps due to software errors [1]. Hence, an embedded system which is a thing on the IoT running complex algorithms, techniques need to be devised for reducing the complexity and the capacity of verification efforts.

An embedded system is comprised of a hardware and a software. The software is a complex piece of code that is prone to errors due to the translation process that converts the high-level code to assembly code. Assembly code is the object code, which is larger in size and complexity as compared to its high-level counterpart. The biggest challenge in the application of formal verification techniques is the large size of the code being executed on the embedded system. Therefore, the success and efficiency of formal verification techniques are highly dependent on the reduction of the size of the code leading to the need for an abstraction technique to minimize the length of the code.

The abstraction technique is a transformation of the object code that will significantly reduce the complexity of the verification process. In this paper, we propose a novel abstraction technique called Static Stuttering Abstraction (SSA). As the name suggests, this technique is applied to the object code directly. The abstraction is developed in the context of refinement-based verification, a formal verification technique. In refinement-based verification, both the specification and the implementation of the system are expressed as Transition Systems (TS). The specification TS is the behavior of the system expressed as states and transitions, whereas the implementation TS is obtained after the software is symbolically simulated at the object code level. The implementation TS is therefore very large as compared to the specification TS, as several transitions of the implementation TS map to a single transition of the specification TS. These several transitions of the implementation TS that map to a single transition of the specification TS are called stuttering transitions. Stuttering transitions usually arise from the execution of stuttering instructions, which are instructions that do not directly modify the state of the system as is visible at the specification level.

The idea with stuttering abstraction is that a finite sequence of stuttering instructions can be merged into one. Such a merger will still preserve the functional behavior of the original implementation TS but will be reduced in size. We call the segment obtained due to the merger an abstracted stuttering segment. Also, we call the reduced TS obtained using such mergers as the abstracted implementation TS. In this paper, we present a methodology to apply abstractions on the stuttering instructions of the implementation TS named static stuttering abstraction (SSA).

The rest of this paper is organized as follows. Background is presented in Section II. Section III details related work. The abstraction technique is described in Section IV. Section V details the case studies and gives verification results. Conclusion and future work are noted in Section VI.

II. Background

In refinement-based formal verification, both the implementation and specification are expressed as a TS. A TS is defined as follows [2].

***Definition** 1:* A TS $\mathcal{M}$ 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 state transition is of the form $\langle w, v\rangle$, where w, $v \in$ S.

MM_S and MM_I denote the specification TS and implementation TS respectively. MM_S is an explicit representation of the requirements of the system thus containing the minimal number of states and transitions. On the other hand, in MM_I a single execution of an instruction in the object code makes up for one or more transitions in MM_I. Therefore, in refinement-based formal verification techniques, the biggest challenge is matching a small set of transitions of MM_S to a large set of transitions of MM_I.

The abstraction technique is developed in the context of Well-Founded Equivalence Bisimulation (WEB) refinement [2]. A key idea in WEB refinement is the notion of refinement maps, which are functions that map implementation states to specification states. The specification TSs are constructed simple and the states include only the predicates relevant to the property being verified. Whereas, the implementation states for object code comprise all the registers in the target micro-controller and memory locations of relevance. Therefore, there is big difference in the abstraction-level of the specification and implementation. Refinement maps essentially extract the relevant variables from the implementation state to construct the corresponding specification state.

The idea with WEB refinement is to match transitions of the implementation with transitions of the specification. Given an implementation transition $\langle w, v\rangle$, checking if this transition matches with a specification transition is achieved by applying the refinement map function $r()$ to both states of the implementation transition. The resulting transition $\langle r(w), r(v)\rangle$ should correspond to a specification transition. However, there are four possibilities. The first possibility is the one mentioned above, that when the refinement map is applied, the implementation transition does correspond to a specification transition. Such an implementation transition is called a non-stuttering transition. The second possibility is that one or both of $r(w)$ and $r(v)$ do not map to any valid specification state. This points to a bug. The third possibility is that both $r(w)$ and $r(v)$ map to the same specification state. In this situation, $\langle w, v\rangle$ is still considered to be a correct transition, but one that is not making visible progress w.r.t. the specification TS. Such a transition is called a stuttering transition. The fourth possibility is that both $r(w)$ and $r(v)$ map to specification states, but $\langle r(w), r(v)\rangle$ does not correspond to any specification transition and $\langle w, v\rangle$ is not a stuttering transition (as described above). The fourth case again corresponds to a bug in the implementation.

The common operations in high-level code are translated to the same set of instructions in the assembly code. Thus, a single set of instructions may have large multiple numbers of occurrences, which correspond to a large piece of the assembly code. The idea of SSA is to identify common multiple occurrences of two or more stuttering instructions named as patterns and reduce the length of the pattern by replacing it with a merger that is a single line instruction but encompasses all the operations performed by the list of instructions in the pattern. Therefore, a pattern comprising of 3 stuttering instructions occurring 100 times in the code will reduce 200 lines of the code and consequently the size of MM_I.

III. Related Work

There are a number of previous approaches that exploit the notion of stuttering to improve verification scalability. An algorithm is presented by Groote and Wijs [3] to check the equivalence between two transition systems based on stuttering. Ray et al. [4] show how to verify concurrent programs using refinement-based on stuttering trace containment. A method for the functional correctness of hardware and low-level software is developed based on refinement-based testing by Jain and Manolios [5]. Stuttering is introduced in the context of probabilistic automata by Delahaye et al. [6]. While the above approaches employ stuttering, they do not apply it to static object code, which is the focus of our work.

Timed Well-Founded Simulation (TWFS) refinement for verification of real-time Field Programmable Gate Array (FPGA) is presented by Jabeen et al. [7]. Reachable states of the FPGA are identified using manually generated invariants, without employing abstractions. This approach is feasible for FPGA only and not for object code with a very large number of instructions as the manual characterization of reachable states for object code is impractical.

Theory of automata is employed to stimulate discrete timed systems and continuous timed systems by Rabinovich [8]. The concept of stuttering is described, but stuttering abstraction is not addressed.

Stuttering invariant properties are expressed using specification languages by Etessami [9] and Dax et al. [10]. The properties distinguish behaviors of systems regardless their stuttering or non-stuttering nature. The properties can be verified using a model checker.

A similar idea of stuttering abstraction is presented by Nejati et al. [11], but static abstraction is not considered.

Stuttering equivalence is employed in the context of the model checking to present an abstraction technique by De-Leon and Grumberg [12]. This abstraction technique is applied dynamically to the transition system and not statically to the object code.

Our abstraction is applicable to recurring patterns of object code instructions that are responsible for millions of transitions in real-time systems. Refinement-based verification, which is a general form of equivalence verification is known to scale well for low-level design artifacts. As can be seen from the experimental results, the object code that constitutes the implementation TS is considerably reduced. Next, we explain SSA.

IV. Automatic Static Stuttering Abstraction

A. Static Stuttering Abstraction (SSA)

The need to develop an abstraction technique for the object code is to ensure efficiency and scalability of the verification process. One of the challenges in refinement-based verification is the complex behavior of the object code. SSA ensures that the object code is transformed into a comparatively smaller piece of code; which consequently reduces the complexity and effort involved in the verification process. SSA is applied after the high-level code is translated to object code, then patterns comprising of stuttering instructions s_i called stuttering patterns s_p with the multiple numbers of occurrences are identified. The pattern is called a stuttering pattern because none of the instructions in the pattern update the values projected by the refinement map. Let's take stepper motor as an example for the specification transition system. The pins of the embedded system (LPC1768) that are responsible for rotating a 4-lead stepper motor and changing the current state is connected to the 4 rightmost pins of general purpose I/O, i.e., LPC_GPIO1. So as long as the assembly instruction does not change the contents of the register associated with LPC GPIO Port 1 (LPC_GPIO1), the instruction is a stuttering instruction s_i. An example of a segment containing a non-stuttering instruction is given below:

```
0x0000067A   2001   MOVS   r0,0x08
0x0000067C   4953   LDR    r1,[pc,332] ; @0x000007CC
0x0000067E   6388   STR    r0,[r1,0x38]
```

The first instruction in the above segment moves the binary 1000 in r0, the second instruction loads the base memory address of the peripheral registers of LPC1768. The third instruction is a store instruction that stores a value of the binary 1000 in rightmost 4 bits of LPC_GPIO1 (to the address calculated by adding an offset of 38 to the base address of 0x000007CC in order to access the address of GPIO Port1 register). Moving a value of 4 in LPC_GPIO1 asserts the leftmost lead of the stepper motor, and hence the stepper motor will take a step and the implementation transition system gets a new state. The STR instruction is hence a non-stuttering instruction n_i, as it has changed the state of the system. This implementation state can be matched to a specification state by employing the refinement map and extracting the rightmost four bits of the register LPC_GPIO1 and mapping them to the specification states. The reason for not abstracting a segment with a non-stuttering instruction n_i is to preserve the independence and behavior of the system, as in a merger a single instruction is supposed to depict multiple parallel operations.

The stuttering patterns s_p on the other hand, comprise of stuttering instructions s_i only. These patterns are observed and are replaced by mergers that preserve the functional behavior of the original MM_I, but will be reduced in size. Below is an example of s_p,

```
0x00000622   4968   LDR    r1, [r3,416]; @0x000007C4
0x00000624   2001   ADDI   r1, 0x04,
0x00000626   6008   STR    r1, [r3,416]
```

The above pattern is a set of instructions that essentially update the contents of a memory location 0x000007C4 by adding 4 to it. Modern processors are not equipped to do such operations in 1 clock cycle. However, we replace these 3 instructions with a single merger as given below,

```
0x00000622   7968125   LAS    [r3,416], 0x04
```

The merger is given a new name LAS abbreviated from Load-Add-Store and is assigned a new opcode for reference. Merger LAS is updating the contents of a memory location 0x000007C4 by directly adding 4 to it in a single instruction. An interesting thing to note here is that the original pattern occupies addresses 0x00000622 to 0x00000626, whereas the merger is only contained at 0x00000622. If each instruction in the original code gives rise to 100 stuttering transitions thus causing a total of 300 stuttering transitions, the merger only corresponds to 100 stuttering transitions. In SSA, we are not concerned with the implementation of the merger on the actual processor. Rather the merger is the abstraction that enables more scalable verification. A library is maintained for stuttering patterns and the corresponding mergers with the type of operation, name of instruction and the opcode shown in Table I.

B. Procedure of SSA

Algorithm 1 shows a procedure that applies SSA to the object code. The inputs to the procedure are,

1) Initial object code file($init\_obj\_code$)
2) A matrix ($path\_opc\_mat$) that contains information regarding opcode of instructions involved in each pattern is shown in Figure 1.

$$M = \begin{bmatrix} LDR, STR & 01001 & 01100 & X & X \\ MOV, STR & 1111000001001111 & 01100 & X & X \\ LSLS, STR & 00000 & 01100 & X & X \\ MOVS, LDR, STR & 00100 & 01001 & 01100 & X \\ MOVS, MOV, STR & 00100 & 1111000001001111 & 01100 & X \\ LDR, LDR_reg, COM, BNE & 01001 & 01101 & 00101 & 11010 \\ LDR, LDR_reg & 01001 & 01101 & X & X \\ TLR, STR & 10000100 & 01100 & X & X \\ LDR, STR(32Bit) & 01001 & 1111100011000001 & X & X \\ LST, TLR & 01110110 & 10000100 & X & X \\ MOV(32Bit), LDR_reg & 1111000001001111 & 01101 & X & X \\ LST, LST & 01110110 & 01110110 & X & X \\ OMS, OMS & 01111001 & 01111001 & X & X \\ TLR, CBNZ & 10000100 & 10111 & X & X \\ OMS, LST & 01111001 & 01110110 & X & X \end{bmatrix}$$

Figure 1. Patterns Opcode Matrix

Each row contains information about the pattern. The first column depicts the instruction types in a pattern. The first row in M contains the pattern LDR-STR, which based on the observation and research has the highest number of frequency in the assembly file. The second column of pattern LDR-STR (row 1) contains the opcode of LDR, the third column contains the opcode of STR. As this pattern only contains 2 instructions so rest of the columns get no opcode values (X). Similarly, row 2 has pattern MOV-STR that has the second highest number of occurrences, second and third columns of row 2 get opcodes values for MOV and STR respectively. Same goes for the rest of the rows and columns.

3) Refinement map (ref_map)

The procedure $Stutt\_Abs$ outputs the updated object code upd_obj_code, which reflects the abstracted implementation TS. The total number of patterns $count$ is calculated statically through a function No_of_Rows (line 2). It is equal to the total number of rows in matrix $patt\_opc\_mat$. N_c keeps a record of the number of patterns that have been abstracted so far in the algorithm. Its initial value is 0 and maximum value must be equal to $count$. Value of N_c will be incremented by one when the search for a pattern starts in $init\_obj\_code$ (line 4). It will be incremented when the search for a pattern in object code completes. s_p is the number of lines in each pattern. It is computed through a function $patt\_size$ (line 6). Function counts the total number of numeric values in each row. Its value must be greater than 2. N_{opc} is a variable that is used to keep track of the number of lines in each pattern (line 7). Function $Next-Ins-Fetcher$ is used to find the Next instruction I_c in $init\_obj\_code$ (line 8). Opc represents the opcode of an instruction I_c and is calculated through function $Opc–Cal$ (line 9).

In order to abstract the instructions, Opc must match with already defined opcode in $patt\_opc\_mat$ (line 10). If both opcodes are matched (line 10), the instruction I_c is stored in $buff$ (line 11) else algorithm will set N_{opc} to 0 (line 26), initialize the buffer $buff$ again, and go to step 6 (line 28) to find the pattern in rest of the object code. To abstract instructions stored in the $buff$, N_{opc} must be equal to s_p (line 12). It indicates that all the required instructions in a pattern are stored in the $buff$. If $s_p \neq buff$, control will go to step 6 again (line 23).

In order to abstract instructions stored in the $buff$, it is required that they all should be stuttering instructions. The stuttering or non-stuttering nature of instructions is computed using a function ref_map (line 13). Output res of function ref_map will be '1' if instructions in the $buff$ are stuttering and '0' in case of non-stuttering instructions. If instructions in the $buff$ are stuttering (line 15), $init\_obj\_code$ is updated by abstracted instruction through a function mrg (lines 16-17). Instructions in $buff$ cannot be abstracted if they are non-stuttering (line 18) and the algorithm will start searching for a pattern in rest of the object code (lines 19-20). obj_code_end is representing the end of an object code. It is computed using function $code\_comp$ (line 27) and the initial value is 0. The algorithm will repeat until each pattern consisting of stuttering instructions is not abstracted in whole object code (line 28). The whole algorithm will repeat until N_c become equal to $count$ (line 29).

The abstracted Object Code depicts the functionality of the original object code and it does not change the essence of the original object code.

```
1:  procedure $Stutt\_Abs(init\_obj\_code, patt\_opc\_mat, ref-map)$
2:      $count = No-of-Rows(patt\_opc\_mat)$
3:      repeat
4:          $N_c{+}{+}$;
5:          repeat
6:              $s_p \leftarrow patt-size(patt\_opc\_mat(N_c, :))$;
7:              $N_{opc}{+}{+}$;
8:              $I_c \leftarrow Next-Ins-Fetcher(init\_obj\_code)$;
9:              $Opc \leftarrow Opc-Cal(I_c)$;
10:             if $[Opc = patt\_opc\_mat(N_c, N_{opc})]$ then
11:                 $buff(N_{opc}) \leftarrow I_c$;
12:                 if $[s_p = N_{opc}]$ then
13:                     $res \leftarrow ref-map(buff)$
14:                     $N\_opc = 0$;
15:                     if $[res = 1]$ then
16:                         $upd\_obj\_code \leftarrow mrg(buff, init\_obj\_code)$;
17:                         $init\_obj\_code \leftarrow upd\_obj\_code$;
18:                     else
19:                         $buff-initialized-again$;
20:                         $again-go-to-step6$;
21:                 else
22:                     $again-go-to-step6$;
23:             else
24:                 $N\_opc = 0$;
25:                 $buff-initialized-again$;
26:                 $again-go-to-step6$;
27:             $obj\_code\_end \leftarrow code\_comp(init\_obj\_code)$;
28:         until $!(obj\_code\_end = 1)$
29:     until $!(N_c = count)$
30: return $upd\_obj\_code$
```

Figure 2. Procedure for Static Stuttering Abstraction

V. Case Study and Results

We have implemented SSA on the object code of an infusion pump. The basic functionality of an infusion pump is to inject medicine, which is done using a stepper motor. This behavior is modeled on an ARM Cortex-M3 based NXP LPC1768 microcontroller, and the assembly code is obtained. The number and type of patterns observed and caught by the automatic SSA are given in Table II. The assembly file is comprised of 335 lines of code for one cycle of execution, which after applying SSA is reduced to 234 instructions.

The result confirms that stuttering abstraction reduces 30.3% of the object code. To show that the algorithm can work consistently on another platform, infusion pump object code was developed for another platform ATMega382P microcontroller. In this case, 26.1% of object code is reduced through SSA. The results in Table II depict that SSA consistently reduces the size of the object code, this is for one execution cycle of the code, whereas in real-time systems the object code is executed in an infinite loop. Also, the reduction in object code will considerably reduce the number of stuttering transitions, which is a huge problem in refinement-based verification.

VI. Conclusion and Future Work

We have developed SSA and shown that the technique can be effectively applied to object code. We have demonstrated static abstractions on object code of infusion pump controller implemented on two different micro controller platforms to reason about the consistency and efficiency of the proposed algorithm. The results demonstrate that static abstraction once applied on stuttering instructions is capable of reducing one-third of the object code, which exponentially reduces the number of stuttering transitions in the implementation transition system. In the context of model checking, several other abstraction techniques have been developed but they have not targeted a very large state space like object code.

In the future, we plan to explore the combination of dynamic stuttering abstraction and static stuttering abstraction and experimentally evaluate this combination. Dynamic stuttering abstraction is the technique where the abstraction is applied to the transition system obtained by symbolically simulating

TABLE I. PATTERNS AND THE MERGERS OF AN LPC1768 OBJECT CODE FOR INFUSION PUMP

Serial Number	**No of Instructions in Pattern**	**Frequency Of Pattern**	**No. of Lines Reduced**	**Instruction Type**	**Instruction Opcode**	**Abstracted Merger Label**	**Merger Opcode (ASCII)**	**Merger Opcode (Binary)**
1	2	13	13	LDR (PC) STR	[01001] [01100]	LST	768384	01110110
2	2	3	3	MOVS STR	[00100] [01100]	MST	778384	01110111
3	2	2	2	LSLS STR	[00000] [01100]	STL	838476	10000110
4	3	15	30	MOVS LDR (PC) STR	[00100] [01001] [01100]	OMS	797783	01111001
5	3	2	4	MOVS MOV (32 Bit) STR	[00100] [F04F] [01100]	VMS	867783	10000110
6	4	9	27	LDR (PC) LDR (REGISTER) CMP BNE	[01001] [00100] [00101] [11010001]	BCL	666776	01100110
7	2	7	7	LDR (PC) LDR (REGISTER)	[01001] [00100]	TLR	847682	10000100
8	3	3	6	LDR (PC) LDR (REGISTER) STR	[01001] [00100] [01100]	RSL	828376	10000010
9	2	2	2	LDR (PC) STR (32 Bit)	[01001] [F8C1]	DLS	687683	01101000
10	2	4	4	LST (User Defined) TLR (User Defined)	[01110110] [10000100]	NLT	787684	01111000
11	2	1	1	MOV (32-Bit) LDR (Register)	[F04F] [00100]	CML	677776	01100111
12	2	2	2	LST (User-Defined) LST (User-Defined)	[01110110] [01110110]	ELT	697684	01101001
13	2	2	1	OMS (User-Defined) OMS (User-Defined)	[01111001] [01111001]	FOS	707983	01110000

TABLE II. RESULTS OBTAINED ON LPC1768 AND ATMEGA382P

Metrics	LPC1768	ATMEGA382P
Number of Lines in Original Object Code	336	524
Number of Lines reduced in Original Object Code	102	139
Number of Lines in Abstracted Object Code	234	385
Total Number of patterns that are abstracted in Object Code	13	21
Percentage of Object Code Abstraction	30.3%	26.5%

the object code.

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] “Medical Device Recalls,” 2017, URL: https://www.fda.gov/MedicalDevices/Safety/ListofRecalls/ucm535289.htm [accessed: Nov,2018].

[2] P. Manolios, “Mechanical verification of reactive systems,” PhD thesis, University of Texas at Austin, 2001.

[3] J. F. Groote and A. Wijs, “An O(m log n) Algorithm for Stuttering Equivalence and Branching Bisimulation,” CoRR, vol. abs/1601.01478, 2016.

[4] S. Ray and R. Sumners, “Specification and Verification of Concurrent Programs Through Refinements,” J. Autom. Reasoning, vol. 51, no. 3, 2013, pp. 241–280.

[5] M. Jain and P. Manolios, “An Efficient Runtime Validation Framework based on the Theory of Refinement,” CoRR, vol. abs/1703.05317, 2017.

[6] B. Delahaye, K. G. Larsen, and A. Legay, “Stuttering for Abstract Probabilistic Automata,” J. Log. Algebr. Program., vol. 83, no. 1, 2014, pp. 1–19.

[7] 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, 2017, pp. 197–203.

[8] A. M. Rabinovich, “Automata over continuous time,” Theor. Comput. Sci., vol. 300, no. 1-3, 2003, pp. 331–363.

[9] 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.

[10] 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, 2009, pp. 244–254.

[11] S. Nejati, A. Gurfinkel, and M. Chechik, “Stuttering Abstraction for Model Checking,” in Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany, 2005, pp. 311–320.

[12] H. De-Leon and O. Grumberg, “Modular Abstractions for Verifying Real-Time Distributed Systems,” Formal Methods in System Designg, vol. 2, 1993, pp. 7–43.