Inconsistencies-based Multi-Region Protocol Verification

Tukaram Muske, Amey Zare
TRDDC, Tata Consultancy Services,
54 B, Hadapsar I.E., Pune, India
{t.muske, amey.zare}@tcs.com

***Abstract*—Software in complex systems like embedded systems usually include protocols (Sleep Wakeup, Controller Area Network Communication, and so on) implemented in multiple code-regions, and these protocols are crucial for the system correctness. For such protocol implementations, code review and testing often fail to detect some of the critical bugs. Many of these bugs are traced back to inconsistencies in the implemented code-regions. We present a new verification technique that identifies likely coding inconsistencies by computing and comparing protocol-critical information over given protocol code-regions. These inconsistencies are then manually validated. In our experiments, the presented technique detected critical bugs that were missed during code reviews and testing.**

***Index Terms*—Embedded Systems; Validation and Verification; Protocols Verification; Coding Inconsistencies**

I. Introduction

Complex systems such as embedded systems usually implement various protocols like as Security, Controller Area Network (CAN) communication and Sleep Wakeup in certain patterns. In one pattern, actions of these protocols are implemented over several parts of the system. For example, actions in CAN communication protocol [1] include data encoding/decoding, message and acknowledgments sending/receiving, error checking, and so on. These actions are often complementary to each other and are implemented in different parts of the code. Further, the actions in a protocol may be required and implemented by multiple components of the embedded system, leading to several similar implementations of the protocol. Henceforth in the paper, a part of code that implements protocol action(s) separately is referred to as a *region*. Thus, such an implemented protocol consists of multiple code regions, and functionality wise, they can be similar or opposite.

Verification of an embedded system is of utmost importance [2]–[4], and it includes proving correctness of intended system functionalities and making sure unintended behaviors are absent [5]. In order to do this, it has become necessary to ensure the included protocols are correctly implemented, as they are crucial for correctness of the system functionality. We describe this by using a Sleep Wakeup protocol that is usually implemented in three regions - Startup, Sleep and Wakeup [6]. This protocol is typically used in battery-powered systems to minimize power consumed by a microcontroller (also referred to as Electronic Control Unit (ECU)). Such power minimization is achieved by toggling the ECU between low power consumption mode (*Sleep* state) and high power consumption mode (*Run* state).

Figure 1 presents a sample implementation of Sleep Wakeup protocol. In this implementation, the Sleep region starts at line 41 and ends at line 45. This region performs certain actions to reduce power consumption before an ECU enters the *Sleep* state. These actions include configuring registers (hardware ports), disabling hardware such as timers, CAN communication channels, and ADC (Analog to Digital Converter). The Startup region (lines 11 to 18) and Wakeup region (lines 83 to 90) perform similar actions before the ECU enters the *Run* state. These actions are opposite to the actions performed in the Sleep region and often include reconfiguring the registers, enabling hardware that were disabled before entering the *Sleep* state.

When a protocol is implemented in multiple regions, possibility of defect introduction increases. This is explained below with respect to the sample implementation.

1) Let us assume that the system functionality (implemented by *perform_Job* function) requires register TRISA (Port A) to be configured with 0x00 value. The Startup region performs the expected register configuration, however the Wakeup region misses to do so. This mismatch can lead to unexpected system behaviors.
2) The CAN communication channel (CANCHNL0) and timer (Timer0) are always enabled in the Startup and Wakeup regions, however their disabling is missed in the Sleep region. Having them enabled during the ECU *Sleep* state can result in more power consumption, and this may lead to battery drain.
3) The hardware HWx is always turned *off* in the Sleep region, but it is missed to turn *on* in the Wakeup region. Due to this, it remains *off* during execution of the system functionality code, and a read or write access to such hardware can result in unexpected *watch dog timer reset* [7].
4) The analog-to-digital converter (ADC) is always disabled in the Sleep region, whereas it is possibly enabled in the Wakeup region. When the Wakeup region fails to enable the ADC, accessing the ADC in the system functionality code can lead to unexpected behaviors.

The above described defects are introduced due to coding inconsistencies, and detecting all such defects in practice

```
 1. int main()
 2. {
     ...
11.  // Startup region begin
12.  TRISA = 0x00; //Port A configuration
13.  CANCHNL0=0x0010; //Enable CAN channel
14.  Timer0 &= 0x80; //Enable Timer 0
15.  HWx = 0x0000; //Switch-on hardware X
16.  ADC = 0x0010; //Enable ADC
17.  var1 = 10;
18.  // Startup region end
19.  ...
20.  while(1)
21.  {
22.   perform_Job();//System functionality
23.   sleep_Wakeup_sequence();
24.  }
25. }
```

```
31. void sleep_Wakeup_sequence(){
     ...
41.  // Sleep region begin
42.  TRISA = 0xff; //Port A configuration
43.  HWx = 0xffff; //Switch-off hardware X
44.  ADC = 0x0001; //Disable ADC
45.  // Sleep region end
46.  SYS_REG = 0x7fff;//ECU in Sleep state
47.  ...
48.  while( !wakeup_condition );
     ...
81.  SYS_REG = 0x8fff; //ECU in Run state
82.  ...
83.  // Wakeup region begin
84.  TRISA = 0xff; //Port A configuration
85.  CANCHNL0=0x0010;//Enable CAN channel
86.  Timer0 &= 0x80; //Enable Timer 0
87.  if(...){
88.    ADC = 0x0010; //Enable ADC
89.  }
90.  // Wakeup region end
91. }
```

Fig. 1. Sample implementation of a Sleep Wakeup protocol

through code reviews may not be possible. This is because, the protocol regions can start and end anywhere in the application and may span over thousands of lines of code that configures (initializes) hundreds of registers (variables). Further, it is not always guaranteed that all such defects will be detected during system testing. For example, the defect (2), due to miss of disabling of CAN channels and timers in the Sleep region, can not be observed via system output parameters. Hence, detecting this defect using testing is difficult as it requires use of sophisticated power consumption monitoring techniques. Due to these issues, a verification technique that helps in automatic and early detection of such defects is always useful.

This paper presents a verification technique that accepts the protocol regions to be verified as inputs and detects possible defects by checking consistency over these regions. This technique is based on computing certain protocol-critical information over each of these regions (referred to as Regional Information), comparing the regional information, and raising an inconsistency so found as a possible defect. Further, this paper shortly describes a framework to compute the required regional information over the input regions. The described framework, first identifies program points that lie inside a given region and later computes the regional information as an effect of the identified program points. This technique to compute regional information is referred as *regional analysis*.

We applied the proposed inconsistencies detection technique to verify Sleep Wakeup protocols in two C applications from automotive industry. The empirical results indicate - a) the presented verification technique detects defects in protocol implementations, which are missed by the other defect finding techniques such as testing and manual code reviews, *and* b) like for any other static analysis technique, generation of false alarms is a concern for our technique.

The key contributions of this paper are - a) an idea to break a complex protocol implementation into similar or opposite regions for the protocol verification, b) a framework for the regional information computation, *and* c) an approach to detect likely inconsistencies by comparing regional information and viewing them as possible defects.

Paper outline: Section II describes the inconsistencies-based verification technique, and Section III provides details of the regional analysis framework. The experiments and their results are described in Section IV. Section V and Section VI, respectively, present related work and conclusion.

II. Multi-Region Protocol Verification

This section describes an inconsistencies-based approach to verify a multi-region protocol implementation by using examples of Sleep Wakeup and CAN communication protocols.

A. Protocol Region: Definition

We define a region starting at P_S and ending at P_E as the part of code having program points that appear on a path originating at P_S and terminating at P_E. The program points in a given region, thus identified, are referred to as *in-region* points, and they include assignment and conditional statements, calls to functions, return statements, etc. Table I provides a few sample regions for a code snippet shown in Figure 2 and their in-region points excluding the region boundaries. It uses line numbers to denote the program points.

As there exists a variety of protocols and each protocol can be implemented in different ways, automatic identification of regions is difficult and may not be generic. Hence, we accept them as inputs specified by their start and end points. In practice, the start and end points of a given region can appear anywhere in the application, and only the code belonging to the given region needs to be analyzed for computation of the intended regional information. For example, in Figure 1, variable ADC is *possibly* modified over the Wakeup region (lines 83 to 90), but computing this modification type over

```
 1. void main()          | 21. void func()
 2. {                    | 22. {
 3.    gVar1 = 1;        | 23.    var1 = 1;
 4.    while(c1)         | 24.    if ( c2 )
 5.    {                 | 25.       var2 = 2;
 6.        gVar2 = 2;    | 26.    else
 7.        func();       | 27.       var3 = 3;
 8.        gVar3 = 3;    | 28.    var4 = 4;
 9.        gVar4 = 4;    | 29.    if ( c3 )
10.    }                 | 30.       var5 = 5;
11.}                     | 31. }
```

Fig. 2. Code snippet for sample regions

TABLE I
SAMPLE REGIONS

Sample Region	Start Point	End Point	In-Region Points
I	6	8	7, [23-30]
II	6	28	7, [23-27]
III	28	6	29, 30, 8, 9, 4
IV	28	23	29, 30, 8, 9, 4, 6, 7
V	25	9	[28-30], 8
VI	25	30	[27-29], [4-9], 23, 24
VII	30	25	[4-9], 23, 24, [27-29]

the complete function (*sleep_Wakeup_sequence*) would find it as *definite*.

B. Sleep Wakeup Protocol Verification

As discussed earlier in Section I, the defects in the sample protocol implementation in Figure 1 arise due to either wrong or miss of a register/variable initialization in one of the regions. In order to detect these defects, a systematic approach is required to select, compute, and compare suitable information over the protocol regions.

1) Regional information computation: We select and compute below regional information to verify a Sleep Wakeup protocol implementation.

i. *Regional Modification Type:* Modification type of a register/variable over the protocol regions is suitable to check if these regions consistently modify the involved register/variable. An inconsistency found this way represents a miss of a register/variable initialization in one of the regions. Computation of such regional modification type of a variable v over an input region starting at R_S and ending at R_E points is described below.
   a) *Definite (D):* The regional modification type of v is *definite* if each path originating at R_S and ending at R_E modifies v.
   b) *Possible (P):* The regional modification type of v is *possible* only if v is modified along at least one path and not by all the paths that originate at R_S and end at R_E.
   c) *No (N):* v has *no* modification type when no path originating at R_S and ending at R_E modifies v.

ii. *Regional Values:* Values assigned to a variable/register over each of the input regions (regional values) are suitable to detect mismatch in the initialization values of the register/variable. Such a mismatch in the regional values over two expected similar regions represents a wrong initialization of the register/variable in any one of the regions.

2) Regional inconsistencies detection: In order to detect the possible defects, the regional information computed over each of the Sleep Wakeup protocol regions is compared as described below.

Comparing Regional Modification Types: Regional modification types of a register/variable are compared in below combinations to detect miss of a register configuration (variable initialization).

a) Startup Vs Wakeup, because a register configuration (variable initialization) in the Startup region should have its corresponding configuration (initialization) in the Wakeup region.
b) Sleep Vs Wakeup, because a register configuration in Sleep region should have its opposite configuration in the Wakeup region. It is to note that, only registers are considered in this combination, and the hardware ports, timers, ADC, other hardware, etc are to be treated as the registers.

Given the comparisons in the aforementioned combinations, it is intuitive that such comparisons are not required for Startup Vs Sleep. In these comparisons, consistency is reported only if both the modification types being compared are *definite*, and all other comparison scenarios are treated as inconsistencies. To be conservative, comparison of two *possible* modification types is treated as an inconsistency, since in this setting, a configuration in one region can not be guaranteed to have its corresponding configuration in the other region. Such a conservative approach may lead to an increased number of inconsistencies and thus a high rate of false alarms.

Comparing Regional Values: Regional values of a register/-variable are compared to detect a wrong configuration/initialization. When the regional values over one region differ with the values from some other similar region, such a scenario is reported as an inconsistency. Also, when the regional values of a register/variable can not be computed statically or are found as interval of values, to be conservative, their comparisons are reported as inconsistencies. Such regional value comparisons are performed only for the Startup Vs Wakeup combination. The Sleep Vs Wakeup combination is considered since the variable/register values are not expected to be same over these regions.

Table II presents results of the consistency checks performed for the protocol implementation in Figure 1. All the defects in the implementation (as described in Section I) are represented by the inconsistencies shown in this table.

The regional information used to detect likely inconsistencies is not limited to modification type and values. One can use other Sleep Wakeup protocol-critical information such as call type (*definite, possible and no*) of instructions that enable/disable the interrupts, and other system calls that acquire and release the locks. Comparing such regional information can help to identify miss on a call of these system calls or instructions.

TABLE II
CONSISTENCY CHECKS OVER SAMPLE REGIONS

Variable /Register	Regional Modification Type			Regional Values			Regional Modification Type		
	Startup	Wakeup	Consistency?	Startup	Wakeup	Consistency?	Sleep	Wakeup	Consistency?
TRISA	D	D	√	0x00	0xff	X	D	D	√
CANCHNL0	D	D	√	0x0010	0x0010	√	N	D	X
Timer0	D	D	√	0x80	0x80	√	N	D	X
HWx	D	N	X	0x0000	-	NA	D	N	X
ADC	D	P	X	0x0010	0x0010	√	D	P	X
var1	D	N	X	10	-	X	N	N	NA

C. CAN Communication Protocol Verification

Selection of the regional information for inconsistencies detection varies as per the protocol. For example, the regional information used to verify an implementation of CAN communication protocol may not be same as it is used in the Sleep Wakeup protocol verification. This is because, a CAN protocol is usually implemented by several components of an automobile embedded system such as *wiper, flasher*, and *body* control unit. Thus, there are repeatative implementations of the same CAN protocol and they ought to be consistent with each other.

The regional information that can be used to verify such implementations may include - a) call type (*definite, possible and no*) of the communication services/APIs [1], b) calling sequence of above services/APIs, c) modification type of parameters of the services/APIs related to message sending, d) read type of parameters of the services/APIs related to message receiving.

It is to note that the presented protocols verification technique is not limited to Sleep Wakeup and CAN communication protocols. Through suitable regional information identified, this technique can be applied to other protocols whose implementation is distributed over multiple similar or opposite regions.

III. REGIONAL ANALYSIS FRAMEWORK

This section briefly describes a framework to compute regional information over a region specified by its start and end points. This computation is achieved in two steps. In the first step, in-region points for a given region are marked (referred to as *region marking*), and in the next step, the in-region points are analyzed to obtain the required regional information.

A. Region Marking

As discussed earlier (in Section II-A), identification and analysis of in-region points is essential to compute the intended regional information. Although, we have defined the in-region points by referring to paths between the region boundaries, computing them this way may not be feasible in practice since the number of paths grows exponentially to number of the conditions. Thus, we use *may* and must reachabilities of a program point from region boundaries, in forward and backward direction, to identify if the program point is an in-region point.

Definition: In-region point- A program point P is an in-region point with respect to a region having R_S and R_E as its start and end points respectively only if both of the following hold true.

1) In forward flow, P is *may* reachable from R_S and it is not *must* reachable from R_E.
2) In backward flow, P is *may* reachable from R_E and it is not *must* reachable from R_S.

Here, the *may* reachability subsumes the *must* reachability. Due to space constraints we avoid detailing the region marking step further.

B. Regional Information Computation

The regional information used in inconsistencies detection can be of several types and varies as per the protocols being verified. The regional modification types and values of the variables are applicable to most of the protocols. Thus, we describe their computation as a representative example of the regional information computation.

1) Computation of regional modification types: Data flow analysis [8] is suitable to compute the *must* and *may* modified variables, and their corresponding data flow formalizations are shown in Table III. For simplicity of the shown formalizations, we have assumed the region code is free of pointers and the region boundaries lie in the same function. These formalizations use results of the region marking to compute the required regional information over the in-region points only. In these formalizations, In_n represents the information flowing in at the start of a node n, while Out_n represents the information flowing out of the exit of the node n. The Gen_n corresponds to the information generated as an effect of the node n.

Using *must* and *may* modified variables over a given region, the regional modification types of the variables can be obtained. The *must* modified variables have the *Definite* modification type. The variables which are *may* but not *must* modified, have *Possible* modification type. A variable that is not *may* modified, has *No* modification type.

2) Computation of regional values: A data flow formalization similar to the formalizations shown in Table III can be used to compute the regional values. Due to space constraints, we avoid providing a separate formalization for regional values computation.

IV. EXPERIMENTAL RESULTS

This section describes various experiments performed to verify the Sleep Wakeup protocols and observations from the results.

TABLE III
DFA FORMALIZATIONS FOR REGIONAL MODIFICATION TYPES

Parameter	***Must* Modified Variables**	***May* Modified Variables**
Initialization (Top)	Set of all variables in the application	$\emptyset$
Meet/Join	Intersection	Union
$In_n =$	$\begin{cases} \emptyset & n \text{ is start of a function} \\ \bigcap\limits_{p \epsilon pred(n)} Out_p & \text{Otherwise} \end{cases}$	$\begin{cases} \emptyset & n \text{ is start of a function} \\ \bigcup\limits_{p \epsilon pred(n)} Out_p & \text{Otherwise} \end{cases}$
$Out_n =$	In_n + Gen_n	In_n + Gen_n
$Gen_n =$	$\begin{cases} Top & n \text{ is an } \textit{out-region} \text{ point} \\ v & n \text{ is an } \textit{in-region} \text{ point and defines } v \\ \emptyset & \text{Otherwise} \end{cases}$	$\begin{cases} v & n \text{ is an } \textit{in-region} \text{ point and defines } v \\ \emptyset & \text{Otherwise} \end{cases}$

We implemented the described Sleep Wakeup protocol verification technique in TCS Embedded Code Analyzer (TCS ECA) [9]. TCS ECA is a static analysis tool to verify C source code. We selected two C applications from automotive industry, one of 56 KLOC representing automobile Body Control Module (BDCM) and another of 40 KLOC representing automobile Battery Control Module (BTCM). The boundaries of the Startup, Sleep, and Wakeup regions from both the applications were provided as inputs to TCS ECA during verification of the protocols. Table IV presents information about each of the regions from the selected protocols. This information includes size of the region, number of variables with the *Definite* modification types (DMTVs), and number of variables with the *Possible* modification type (PMTVs).

TABLE IV
REGIONAL ANALYSIS RESULTS

Application	Region	LOC	DMTVs	PMTVs
BDCM	Startup	3168	302	283
	Sleep	838	32	65
	Wakeup	3193	299	288
BTCM	Startup	2246	59	150
	Sleep	1150	62	18
	Wakeup	2246	59	150

TABLE V
COUNTS OF REGIONAL (IN)CONSISTENCIES

Appli-cation	Startup Vs Wakeup				Sleep Vs Wakeup	
	RMTIs	RMTCs	RVIs	RVCs	Register RMTIs	Register RMTCs
BDCM	413	186	143	544	66	33
BTCM	150	59	48	161	48	32

A. Observations from protocols verification

Table V presents the summary of the verification results of the selected Sleep Wakeup protocols. In this table, RMTCs (RMTIs) denotes count of the regional modification type consistencies (inconsistencies), and the RVCs (RVIs) denotes count of the regional values consistencies (inconsistencies).

BDCM Application: Large number of inconsistencies were reported for this application due to *possible* modification types, and its reason was traced to the conditional calls of the functions that initialized the registers/variables in the Wakeup region. Manual review of all the reported inconsistencies, performed by the system developers, took around two hours of manual efforts. Few observations from this activity are mentioned below.

- The review of the RMTIs in Startup Vs Wakeup combination revealed *possible* miss of configuration of four hardware pins in the Startup region. This was due to conditional call of the function that configured the hardware pins. Also, review of these inconsistencies indicated that initializations to two global variables were *definitely* missed in the Wakeup region, and each miss was found to be a coding defect.
- One inconsistency among the reported 86 register RMTIs in Sleep Vs Wakeup combination indicated presence of critical defect, which was due to miss of disabling of the DMA controller in the Sleep region.
- None of the regional values inconsistency in Startup Vs Wakeup represented a coding defect.

BTCM Application: The Startup and Wakeup regions in BTCM were found to be overlapping, hence the inconsistencies reported for this combination were not manually reviewed. On manual review, none of the register modification inconsistency in Sleep Vs Wakeup represented a coding defect. This manual review took around 20 minutes.

B. Other Observations

Inconsistencies-based verification Vs Manual Code Review: We performed an experiment to check effectiveness of the presented verification technique against the manual code review. In this experiment, a developer manually reviewed the protocol code in BDCM application to identify the defects. After six hours of reviewing efforts, the developer was able to identify only one defect related to the miss of disabling of DMA controller, and the other defects were not found during the review. This experiment indicated the presented technique is useful in detecting more bugs which are usually missed during code reviews.

Inconsistencies-based verification Vs Testing: Both the selected applications were after their testing at unit and system levels. The testing of the selected BDCM application was unable to detect the defects that were detected by the inconsistencies-based verification approach. It indicated the effectiveness of the presented verification technique in detecting defects which are hard to find during testing.

Impact of conservative approach: We performed few experiments to observe impact of the conservative approach taken during the computation of inconsistencies. These experiments indicated that around 55% of the reported inconsistencies were due to treating a comparison of two *possible* modification types as an inconsistency. In our experiments, although the inconsistencies due to conservative approach did not contribute in defects identification, we believe such an approach may benefit on some other applications. Further, these experiments indicate that the conservative approach can be avoided in order to generate fewer false alarms at the cost of miss of detection some defects.

V. Related Work

Coding inconsistencies have been used earlier for bugs detection. Engler et al. [10] used automated rule extraction to get the programmer beliefs, and one of the contradictory beliefs are treated as an error. Lu et al. [11] have used an inconsistency in updates to the correlated variables for semantic bugs detection. To the best of our knowledge, such coding inconsistencies detection has not been used in the verification of protocol implementations. In our presented technique, the information with which the inconsistencies are computed is critical to the protocol functionality, and it is based on the (dis)similarity of the actions implemented by protocol regions. This regional information is different from the information used by the existing techniques [10][11].

There are a number of protocols corresponding to security, communications, cryptography (data encryption), routing in networks, etc, and many approaches have been proposed to verify their implementations. These approaches use a variety of techniques such as predicate abstraction [12], patterns-based verification [13], model checking, heuristic search, or their combinations [14]. The approaches used and categories of the bugs detected by these techniques are protocol-specific. None of these techniques break a complex protocol implementation into the similar or opposite functionality regions and achieve the protocol verification.

Regional analysis has been used mostly earlier for effective memory management [15][16] and efficient solving of the data flow analysis [17], but it has been rarely used in protocol verifications. In these existing techniques, the regions to be analyzed are automatically identified, where the region boundaries belonged to the same function. Our presented regional analysis framework analyzes a given region whose boundaries can appear anywhere in the application.

VI. Conclusion and Future Work

An idea to break a complex implementation of a protocol into similar or opposite regions and an approach for their verification was presented in this paper. Detecting inconsistencies over the multiple regions of a protocol is an effective verification technique, since there could be a mismatch in their implementations due to coding by multiple developers and its multi-place distribution. Further, discovering such an inconsistency and its associated defect may not be easy using manual reviews and/or conventional testing techniques. Similar have been our observations during the experiments, which indicated usefulness of the presented technique in detection of the critical defects.

The thorough manual review of the reported inconsistencies increased our confidence about correctness of the protocol implementation. This process acted as a systematic review of the implementations, which would have not been possible otherwise. Although the experiments are performed on Sleep Wakeup protocols in embedded domain applications coded in C, we expect similar benefits on other domain/language protocols too, due to common coding practices.

Like any other static analysis technique, our experiments depicted a very high rate of false alarms (around 98%) for the presented verification technique. We plan to work on minimizing falsely reported inconsistencies in the near future.

References

[1] AUTOSAR, "Autosar specification of communication v2.0.1," Jun. 2006.

[2] J. C. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 547–550.

[3] D. R. Wallace and R. U. Fujii, "Software verification and validation: an overview," *Software, IEEE*, no. 3, pp. 10–17, 1989.

[4] J. Yoo, E. Jee, and S. Cha, "Formal modeling and verification of safety-critical software," *Software, IEEE*, no. 3, pp. 42–49, 2009.

[5] P. Cousot and R. Cousot, "Verification of embedded software: Problems and perspectives," in *Embedded Software*. Springer, 2001, pp. 97–113.

[6] AUTOSAR, "Autosar specification of ecu state manager v3.0.0 r4.0 rev 3," Nov. 2011.

[7] J. Santic, "Watchdog timer techniques," *Embedded Systems Programming*, vol. 8, no. 4, pp. 58–69, 1995.

[8] U. Khedker, A. Sanyal, and B. Sathe, *Data Flow Analysis: Theory and Practice*. Taylor & Francis, 2009.

[9] TCS Embedded Code Analyzer (TCS ECA), http://www.tcs.com/offerings/engineering_services/Pages/TCS-Embedded-Code-Analyzer.aspx, [Online; accessed 25-Aug-2014].

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001.

[11] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 103–116, Oct. 2007.

[12] E. Pek and N. Bogunovic, "Predicate abstraction in protocol verification," in *Telecommunications, 2005. ConTEL 2005. Proceedings of the 8th International Conference on*, vol. 2. IEEE, 2005, pp. 627–632.

[13] L. Bozga, Y. Lakhnech, and M. Périn, "Pattern-based abstraction for verifying secrecy in protocols," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 299–314.

[14] S. Edelkamp, A. L. Lafuente, and S. Leue, *Protocol verification with heuristic search*. Bibliothek der Universität Konstanz, 2001.

[15] S. Cherem and R. Rugina, "Region analysis and transformation for java programs," in *Proceedings of the 4th international symposium on Memory management*. ACM, 2004, pp. 85–96.

[16] R. Rugina, "Region analysis for imperative languages," Cornell University, Tech. Rep., 2003.

[17] Y.-F. Lee, B. G. Ryder, and M. E. Fiuczynski, "Region analysis: A parallel elimination method for data flow analysis," *Software Engineering, IEEE Transactions on*, vol. 21, no. 11, pp. 913–926, 1995.