Static Program Analysis of Embedded Executable Assembly Code

Abstract

We consider the problem of automatically checking if coding standards have been followed in the development of embedded applications. The problem arises from practical considerations because DSP chip manufacturers (in our case Texas Instruments) want various third party software developers to adhere to a certain coding standard to facilitate system integration during application development. Checking for compliance with coding standards, in general, is undecidable. Moreover, only machine code of the system components is available since for proprietary reasons vendors of various components do not want to share their source code. In this paper, we describe an approach based on static analysis of embedded assembly code to check for compliance with such coding standards. This static analysis rests on an abstract interpretation framework. We illustrate our approach by showing how we statically analyze the presence of *hard-coded pointer* variables in embedded assembly code. Hard coded pointer variables are those that are assigned a fixed memory address by the programmer instead of being assigned a value via proper operations in the source language (e.g., *malloc/calloc/realloc* and & operator in C). Our analyzer takes object code as input, disassembles it, builds the flow-graph, and statically analyzes the flow-graph for the presence of dereferenced pointers that are hard coded. The analyzer is currently being extended to check for compliance with other rules adopted by TI as part of its coding standards.

1 Introduction

The Texas Instruments' (TI) *Express DSP Algorithm Interoperability Standard* (XDAIS) [11] defines a set of requirements for DSP code (for TI TMS320 family of DSP processors) that, if followed, will allow system integrators to quickly assemble production quality embedded systems from one or more subsystems. The standard aims to provide a framework that will enable the development of a rich set of Commercial Off-The-Shelf (COTS) marketplace for the DSP components technology for the the TI TMS320 family of DSP chips. This will significantly reduce the time to market for new DSP based products and that will encourage reuse. There are 34 rules and 15 guidelines defined by the standard that program code developed by a third party DSP software vendor should follow. Rules 1 through 6 in the standard fall under the category of "general programming rules" and pertain to standards that should be followed during program coding. Tools are available from TI that check for the compatibility of a program code with most of the non-programming rules (i.e., rules other than 1 through 6).

The non-programming rules are relatively straightforward and hence easy to check automatically (e.g., rule 11 states that all modules must have an *initialization* and *finalization* method). Compliance of a program code with general programming rules, in contrast, is much harder. In fact, this compliance checking is undecidable in general (this constitutes the reason why the DSP industry has found it hard to develop tool for checking compliance of a program code with rules 2 through 6 [11]). As a result, no tool has been developed for checking compliance of a program code with rules 1 through 6. We propose to use static analysis to perform this compliance check. However, static analysis of the program code is complicated by the fact that the source code of the program is not available—vendors generally just ship their binaries that can be linked with other codes. Thus, to check for compliance assembly code has to be analyzed. However, the source code is usually written in C; this information, coupled with our knowledge of how the TI C compiler compiles and assembles code, helps in the analysis of the assembly code.

This paper describes our approach to checking a program's binary code for compatibility with the "general programming rules" defined by the standard. Our approach is based on statically analyzing the disassembled code. Static analysis of assembly code is quite hard, as no type information is available. Thus, for example, distinguishing a pointer variable from a data value becomes quite difficult. Also, most compilers take instruction level parallelism and instruction pipelining provided by modern processors into account while generating code. This further exacerbates the automatic static analysis of assembly code.

Our static analysis framework is based on abstract interpretation. Thus, assembly code is abstractly interpreted (taking instruction level parallelism and pipelining into account) to infer program properties. A *backward analysis* is used since in most cases data type of a memory location has to be inferred by how the value it stores is used. Once a point of use is determined, the analysis proceeds backwards to check for the desired property.

We illustrate our approach by considering how we statically analyze the presence of *hard-coded pointers* (rule 3 of the TI XDAIS [11] standard), i.e., how we check whether a pointer variable has been assigned a constant value by the programmer. We give details of the tool that we have built for this purpose. Other rules can be checked in a similar manner. Note that our goal in this project has been to produce an analyzer that can be used to check for compliance of large commercial quality program codes. Thus, all (nasty) features of C that may impact the analysis have been considered. For example, in hard-codedness analysis, we have to consider cases where pointers are implicitly obtained via array declarations, pointers with double or more levels of indirections (e.g., int **p), pointers to statically allocated global data area, etc.

The main contribution of this paper is to show that embedded machine code can be successfully analyzed via abstract interpretation based techniques, even in the presence of instruction level parallelism and pipelining. We show that abstract interpretation based static analysis can be successfully used to check compliance with coding standards for improving code reuse and interoperability. Indeed, our work has resulted in a practical tool for doing hard-codedness analysis of embedded DSP software.

We assume that the reader is familiar with abstract interpretation; tutorial introduction can be found in chapter 1 of [15].

1.1 General Programming Rules

The coding standard rules, published by TI for software vendors of its DSP chips, that fall under the category of “general programming rules” [11] are the following:

1. All programs must follow the runtime conventions imposed by TI’s implementation of the C programming language.

2. All programs must be reentrant within a preemptive environment including time sliced preemption.

3. All data references must be fully relocatable (subject to alignment requirements). That is, there must be no “hard coded” data memory locations.

4. The code must be fully relocatable. That is, there can be no hard coded program memory locations.

5. Programs must characterize their ROM-ability; i.e., state whether they are ROM-able or not. ROM-ability means that if part of the executable is placed in the DSP ROM, it would still function; this restricts the way global data can be accessed, etc. [11].

6. Programs must never directly access any peripheral device. This includes but is not limited to on-chip DMA’s, timers, I/O devices, and cache control registers.

There are a number of advantages to DSP software vendors writing programs that comply with the published standards. Compliance to standards (i) allows system integrators to easily migrate between TI DSP chips; (ii) enable host tools to simplify a system integrators tasks, including configuration, performance modeling, standard conformance, and debugging; (iii) subsystems from multiple software vendors can be integrated into a single system; (iv) programs are framework-agnostic, that is, they are reusable: the same program can be efficiently used in virtually any application or framework; and, (v) programs can be deployed in purely static as well as dynamic run-time environments (due to code relocatability).

2 Analysis of Hard-coded Pointers

We illustrate our static analysis based approach to compliance checking by showing how we check compliance for rule #3, which states that there should be no hard-coded data memory locations. A data memory locations is hard coded in the assembly code if a constant is moved into a register $\mathtt{R}_i$, and $\mathtt{R}_i$ is then used as a base register in a later instruction. The constant value may of course be transferred to another register $\mathtt{R}_j$ directly or indirectly, and then $\mathtt{R}_j$ used later in dereferencing. Since most of the TI’s DSP code is written in C, data memory locations can be hard coded either in assembly code embedded in a C program, or by using pointers provided in the C language.

Thus, the problem of detecting hard coded references is to check whether a pointer variable that is assigned a constant value is dereferenced or not in the program. Thus, using the 'C' syntax for illustration purposes, given a variable `p` of type `'(int *)'`, we want to check if there is an execution path between a statement of the type `p = k`, where `k` is an expression that yields a constant value, and a later statement containing `*p` (that dereferences a pointer). Of course, the dereferencing may take place directly or indirectly, i.e., we might have an intervening statement `(int *)q = p` followed by a later statement containing `*q`.

Note that if a program only hard codes a pointer variable but never dereferences it, the program is deemed safe. It is only after such a pointer variable is dereferenced during subsequent execution, that the program is deemed unsafe.

Clearly, the problem of detecting hard coded references is undecidable in general [51]. So we employ static analysis for detection of hard codedness (from this point on, we'll call the analysis *hard-codedness analysis*). Our hard-codedness analysis is conservative in that if it declares a data memory location to be hard coded (HC), we are certain that it is hard coded. However, if it says that a data memory location is not hard coded (NHC) then we cannot be sure—it may be hard coded or it may not be hard coded, we just don't have enough information.[1] We could also organize our analysis so that if it determines a location to be NHC, then we can be sure that indeed it is NHC; if it determines a location to be HC, then we are not certain and it could be either. However, the latter kind of analysis will detect very few pointers to be NHC, since in a language like C there are far too many ways in which memory locations can be aliased and hard coded indirectly.

Note also that the problem of detecting dereferencing of hard-coded pointers subsumes the problem of detecting dereferencing of NULL pointers. This is because a NULL pointer is a pointer that has been assigned a special constant (usually 0x0). Thus our analysis will also detect NULL pointer dereferences. Similarly, hard-codedness analysis subsumes analysis for checking if un-initialized pointer variables are dereferenced. This is because hard-codedness analysis attempts to check if a pointer dereference is reachable from a point of initialization; and thus will detect any pointers that are dereferenced but not initialized. Thus, our hard-codedness analysis performs two of the checks proposed by the UNO project [53] at the assembly level. The UNO project claims that NULL pointers, un-initialized pointers, and array out of bounds reference are three most common run-time programming errors.

When analyzing at the assembly level, all the type information is unavailable, and distinguishing between constants stored in integer variables from constant addresses stored in pointer variables is hard. The only way to distinguish between the two is to check if a register is dereferenced or used as a base register at some program point, and if so, we can go backwards from that program point and check to see if this register was directly or indirectly assigned a constant value.

We use an abstract interpretation [14, 15] based framework for static analysis. The abstract

[1]Our experience running the analyzer we have developed (described later), however, shows that in most cases an HC or an NHC determination can be made with nearly 100% certainty.

domain is quite simple and consists of four values: $\perp$, $\top$, HC and NHC. Abstract operators are defined for pointer arithmetic based on this abstract domains, and the abstract values propagated in the flowgraph of the program (obtained by disassembling the machine code, and analyzing the control flow). The abstraction is shown to be safe (by constructing a Galois connection [14, 15]). Abstract semantics of the program are defined via recursive equations. The "collecting semantics" of the flow-graph is then computed via fix-points, which then allows us to check for hard codedness.

A static analyzer has been implemented and used for analyzing a suite of DSP program codes obtained from Texas Instruments. Performance results on this suite of programs are reported. For most programs, our system is able to detect occurrences of hard-codedness with good accuracy. This is primarily because most practical DSP program codes use pointers in non-convoluted ways, and our static analyzer is able to detect such cases with good precision.

3 Abstract Interpretation based Static Program Analysis

Static program analysis (or static analysis for brevity) is defined as any analysis of a program carried out without completely executing the program. Static analysis provides significant benefits and is increasingly recognized as a fundamental tool for analyzing programs. The traditional data-flow analysis found in compiler back-ends is an example of static analysis [13]. Another example of static analysis is *abstract interpretation*, in which a program's data and operations are approximated and the program *abstractly executed* (abstraction is done in a way to ensure termination) to collect information [14, 15].

In abstract interpretation based static analysis, domains from which variables draw their values are approximated by *abstract domains.* The original domain is called a concrete domain. Further, for each operation over these domains, a corresponding abstract operation is defined over the abstract domain. A program is represented by a flow chart, and its semantics is given via a mapping from arcs that connect nodes of the flow chart to environments, where an environment is a mapping from variables to values in the concrete domain. In the abstract semantics, the environment is abstracted as a mapping from variables to values in the abstract domain. A state is defined as a pair $\langle arc,\ env\rangle$ where *arc* is an arc in the flow chart and *env* is the environment that exists along that arc at a given moment. The same arc may be in different states (depending on the values of the variables), at different moments during the execution. The function *next* maps a given state to the next state that will be reached in the execution. The meaning of the program P is solution to the recursive equation:

$$P = next \bullet P$$

which is given by:

$$fix(\lambda f.next \bullet f)$$

In the abstract interpretation framework [14], a collecting semantics is used, i.e., we consider the set of all the abstract environments that might be associated with a program point (an arc in

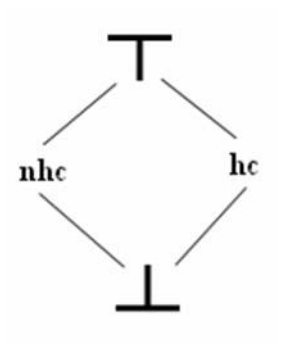

Figure 1: **Lattice Abstraction**

the flowgraph). This set of abstract environments is called a context. The collecting semantics thus associates a *context* with each arc. A context is a member of the powerset of the set of all environments.

$$Contexts = 2^{Env}$$

The context associated with a particular arc is the set of all environments that can exist along that arc, if we started execution from any of the initial nodes of the flow graph.

The collecting semantics is not computable because it gives exact information, it is therefore approximated. Given a pointer variable that is dereferenced, we are only interested in whether this pointer was hard coded earlier or not. Let $\mathcal{A}$ be the set of all memory addresses. An environment maps a pointer variable to an address in the set $\mathcal{A}$ (for hard codedness analysis we are only interested in pointer variables). $\mathcal{A}$ can be divided into two sets $\mathcal{A}_{nhc}$ and $\mathcal{A}_{hc}$, where $\mathcal{A}_{nhc}$ represents legitimate (i.e., not hard coded) memory addresses (those that might be returned by systems calls *malloc*, *calloc* or *realloc*, or returned by address-of operation, e.g., & operator in C), and $\mathcal{A}_{hc}$ represents the rest of memory addresses. Thus, $\mathcal{A}_{nhc} \cup \mathcal{A}_{hc} = \mathcal{A}$ and $\mathcal{A}_{nhc} \cap \mathcal{A}_{hc} = \phi$. We approximate the domain of addresses by an abstract domain, $\mathcal{A}_\alpha$ where $\mathcal{A}_\alpha = \{\perp, hc, nhc, \top\}$. Note that *hc* abstract the elements in the set $\mathcal{A}_{hc}$ while *nhc* abstracts the elements of the set $\mathcal{A}_{nhc}$; the value $\perp$ represents complete lack of information, while $\top$ denotes that we cannot decide whether the value is *hc* or *nhc*. $\mathcal{A}_\alpha$ forms a lattice as shown in Figure 1.

Note that we assume that NHC addresses are those that are derived from calls to memory allocation routines (malloc, etc.), and we also assume that any offset from an NHC address is also NHC. Thus, the sets $\mathcal{A}_{hc}$ and $\mathcal{A}_{nhc}$ are determined by each program and may vary from execution to execution depending on where the heap and stack are allocated. Our analysis is able to cope with this, since it regards any address derived from *malloc, calloc, realloc* and the & operator to be safe, while any address directly assigned in the program is regarded as unsafe.

Following the abstract interpretation approach, we define the abstraction and the concretization functions, α and γ, respectively.
α : *Contexts* $\rightarrow$ *Abstract_Contexts*, where *Abstract_Contexts* consists of abstract environments which map pointer variables to values in $\mathcal{A}_\alpha$.

$$\begin{aligned}\alpha(C) &= \bot,\ C = \{\};\\ &= nhc,\ C \subseteq \mathcal{A}_{nhc};\\ &= hc,\ C \subseteq \mathcal{A}_{hc};\\ &= \top\ \ otherwise;\end{aligned}$$

$\gamma : Abstract\_Contexts \rightarrow Contexts$

$$\begin{aligned}\gamma(S) &= \{\},\ S = \bot,;\\ &= \mathcal{A}_{hc},\ S = hc;\\ &= \mathcal{A}_{nhc},\ S = nhc;\\ &= \mathcal{A},\ otherwise\end{aligned}$$

It is easy to see that α and γ constitute a Galois connection [14], i.e., $x = \alpha(\gamma(x))$ and $\gamma(\alpha(y)) \supseteq y$. The existence of a Galois connection guarantees that our analysis is sound.

We next have to abstract the operators involving pointers(Table 1). Pointers can be involved in pointer arithmetic expressions that use `+` and `-` operations. Given a statement `p = q + i` where `p` and `q` are pointers to integers and `i` an integer, then if `q` is hard-coded, our analysis should infer that `p` is also hard-coded. Likewise, if `q` is NHC, our analysis should infer that `p` is also NHC. However, given pointers `p, q`, and `r`, and `p = q + r`, then for `p` to be inferred as hard-coded, both `q` and `r` must be hard-coded. If, say, `q` is hard coded but `r` is not, then `q` must be treated as an offset from the safe pointer `r`, and our analysis should infer that `p` is not hard coded. With this in mind the definition of abstracted `+` and `-` operations for pointers is shown in the table below:

+/-	hc	nhc	$\bot$	$\top$
hc	hc	nhc	$\bot$	$\top$
nhc	nhc	nhc	nhc	nhc
$\bot$	$\bot$	nhc	$\bot$	$\bot$
$\top$	$\top$	nhc	$\bot$	$\top$

Table 1: Pointer Arithmetic

Once the abstract operators are defined, we can compute the abstract semantics of the program by computing the fix point of the recursive equation:

$$P_\alpha = next \bullet P_\alpha$$

where P_α is the abstract program (the abstract program ignores instructions that are determined to not affect a pointer value). Note that the abstract contexts (*Abstract_Contexts*) forms a lattice under the subset relation, $\subseteq$, where the join ($\sqcup$) and meet ($\sqcap$) operations correspond to the set union and set intersection operations respectively. The join operation is used to merge the abstract contexts of multiple arcs leading into a common node. The abstract semantics once computed, tells us which pointer references are hard coded.

Finally, we have to show that our analysis is sound. The soundness of the analysis follows if we can show that α and γ are mutually consistent and that *Abstract_Contexts* form a lattice.

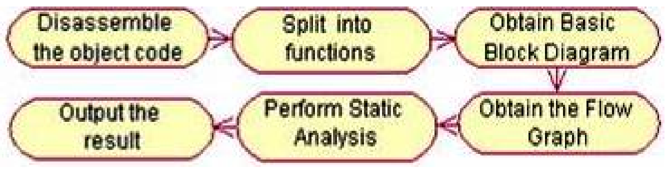

Figure 2: **Activity Diagram for our tool**

Theorem: The Hard-codedness Analysis formulated above is sound.
Proof Sketch: Consistency of α and γ functions is established by showing that they constitute a *Galois connection* [14]. That is:

$$x = \alpha(\gamma(x)) \qquad ...(1)$$

$$\gamma(\alpha(y)) \supseteq y \qquad ...(2)$$

It is easy to check that both (1) and (2) hold. Given the way *Abstract_Contexts* are defined, it is also easy to see that the set of *Abstract_Contexts* forms a lattice under the subset ($\subseteq$) relation. Thus, our analysis is indeed sound. $\square$

4 The Analysis Algorithm

4.1 Overview

As discussed earlier, the analyzer has access only to the object (binary) code which is to be checked for compliance with the standard. So, the given object code is disassembled and the corresponding *assembly language code* is obtained. The disassembly is performed using *TI Code Composer Studio* [9, 10]. The disassembled code is provided as input to the static analyzer which produces a result which indicates whether the code is compliant with the rule.

To check for compliance with the standard, the binary code that is given as input is never executed. The analyzer scans through the disassembled code statically and checks whether there are any hard coded addresses. *The basic aim of the analysis is to find a path from the point in which the dereferencing of a pointer occurs to the point at which an address is assigned to the pointer and then check whether that address is legitimate or not.*

In the following discussion when we use the term *"safe"* it means that the program has no hard coded addresses. We use the term *"unsafe"* to mean the opposite.

Figure 2 shows the various steps involved in the analyzer we have developed. After the disassembly of object code, the assembly code is split into functions. The analysis is first done function by function (corresponds to individual functions in the source code). For each function, the analyzer computes its basic blocks and constructs the flow graph. The flow graph is then statically analyzed. Once information for various functions (including `main`) has been collected, an interprocedural analysis is carried out, to analyze the entire program. The functions may have to be re-analyzed (until a fix-point is reached) during this interprocedural analysis.

It should be noted that there are advantages as well as disadvantages of performing static analysis at the assembly level. W.r.t. advantages, note that (i) parsing of source code is replaced by parsing of assembly instructions, which is considerably easier; (ii) in the source code pointers may occur in complex expressions (e.g., a pointer in a struct inside another struct), which makes the analysis complex at the source code level; at the assembly level all pointers manipulations are handled via registers, and thus all occurrences of pointers appear very similar, regardless of how they were expressed in the source code; (iii) since we analyze assembly code, which is source language independent, many of the nasty features of C get handled by the compiler and appear in considerably sanitized form in the assembly code.

W.r.t. disadvantages, (i) all type information is lost at the assembly level, so addresses are indistinguishable from data, thus abstracting information becomes harder, complicating the analysis. (ii) Registers are repeatedly reused, for holding values of different variables, and thus a lot of analysis is involved in figuring out which occurrences of a register in various instructions refers to the same value; (iii) Assembly code is much more verbose than source code, as well as it heavily employs pipelining and instruction level parallelism; thus, a very good understanding of the processor architecture is needed which needs to be then modeled in the abstract semantics. As an example, consider the following C code:

```
void main(){
 int a=1, b=2, c=3;
}
```

which translates to the assembly code shown below:

```
000007A0 main:
000007A0 07BE09C2 SUB.D2 SP,0x10,SP
000007A4 02002042 MVK.D2 1,B4
000007A8 0200012B MVK.S2 0x0002,B4
000007AC 023C22F6 || STW.D2T2 B4,*+SP[0x1]
000007B0 020001AB MVK.S2 0x0003,B4
000007B4 023C42F6 || STW.D2T2 B4,*+SP[0x2]
000007B8 023C62F6 STW.D2T2 B4,*+SP[0x3]
000007BC 00002000 NOP 2
000007C0 008C8362 BNOP.S2 B3,4
000007C4 07800852 ADDK.S2 16,SP
000007C8 00000000 NOP
000007CC 00000000 NOP
```

The presence of the || characters denotes instruction level parallelism. We can see the usage of register B4 in line 5 and a value assigned to it in the line 4. But, the content of B4 used in line 5 is not the one assigned in line 4. This is due to the presence of parallelism. Note that our analyzer takes care of parallelism and instruction pipelining while computing the abstract semantics.

4.2 Phases in the Analysis

The analyzer functions in two phases. In the first phase, it scans through the flowgraph and detects all the register dereferencing that correspond to the dereferencing of pointer variables in the source code. It stores this information in the various basic blocks in a set. We call such a set an *unsafe* set. The unsafe sets represent the abstract contexts discussed earlier. There is an unsafe set for each pointer that is dereferenced. This unsafe set records all the registers that may potentially hold the address corresponding to this pointer. In the second phase the unsafe sets are iteratively refined, until a fixpoint is reached.

Phase1: Detecting dereferencing of pointers: To detect the dereferencing of pointers, the analyzer starts from the entry nodes in the flow-graph and visits every reachable node in the flow-graph. While visiting any node in the flow-graph, it checks for the occurrences of pointer dereferencing. Dereferencing of a pointer is detected in the disassembled code when a register other than the stack pointer(SP) is used as the base register. Dereferenced registers are recorded in the unsafe sets. There is an unsafe set for each dereferencing operation in the program.

Phase 2: Checking if dereferencing is safe: In the second phase, the analyzer uses the information gathered in the first phase and ascertains the safety of each of the unsafe sets. This is done by analyzing the safety of the pointer across all possible paths through which the pointer might have got its value at the point of its dereferencing. That is, if there are multiple locations at which the same pointer may be hard coded (which may correspond to multiple paths in the flow graph), the analyzer will be able detect and report all such locations.

Refining Unsafe Sets: The unsafe sets are built iteratively, via a fix point computation, as described earlier in the presentation of abstract interpretation framework. Initially, the unsafe set is empty; once phase 1 detects the dereferencing of a register, it adds that register to the set. So, if register "Reg" is seen as being dereferenced, it is added into the unsafe set, which will now appear as {Reg}.

In phase 2, the analyzer looks for statements in which an element from the unsafe set (in this case *"Reg"*) is used as the destination register. That is, the analyzer is trying to find what is the most recent value that was assigned to the register *"Reg"*. When it detects such an occurrence, say, which corresponds to a statement like *"Reg = Reg1 + Reg2"*, it *deletes* the element *"Reg"* from the unsafe set and *inserts* the elements *"Reg1"* and *"Reg2"* into the unsafe set. Now the unsafe set becomes {*Reg1,Reg2*} .

In phase 2, the analyzer continues with the current unsafe set (looking for the occurrence of both Reg1 and Reg2 as the destination registers in this case). Phase 2 terminates when the status of each of the unsafe set has been determined. As discussed earlier, the status of an unsafe set is deemed to be HC if all the elements in a given unsafe set are hard coded. If at least one of the elements in the unsafe set is not hard-coded, then the corresponding pointer is safe.

Note that if no pointers are dereferenced, the analyzer does not even enter phase 2. In phase 1,

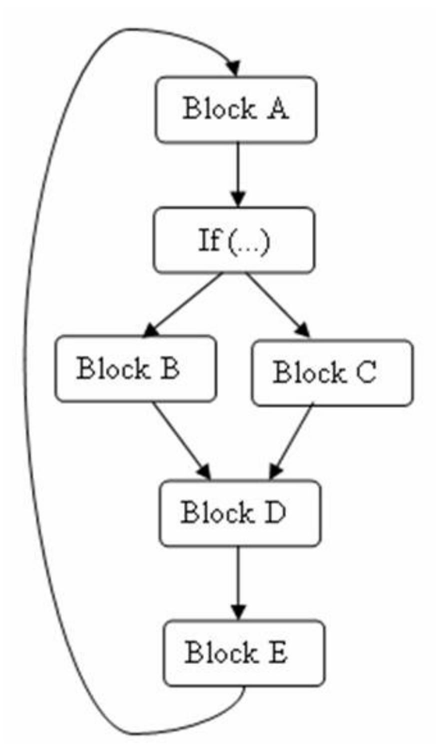

Figure 3: **Need for Merging Unsafe Sets**

the analyzer inspects each line of the assembly code only once. This is achieved by maintaining a set of unsafe sets (SOUS) which is carried through as various instructions in the flow graph.

Merging Information: During phase 2 of the analysis, the analyzer builds and populates the unsafe set. Consider figure 3 which represents part of some control flow graph. If a basic block has multiple successors, then the instructions of that basic block will be analyzed after merging of all the SOUS of all the successors. If the analyzer does not perform this merging operation then the analysis will be of exponential complexity. This is true especially if loops are also involved. The unsafe sets (abstract contexts) form a lattice under subset ordering, the merging involves a join ($\sqcup$) operation, which is simply a set union operation in this case. Static Analysis based approximation and merging of information makes the complexity of the analysis $O(n * m)$ where n is the measure of the size of the flowgraph, and m a measure of the size of (finite) lattices involved. Merging results in information loss. That is, since the common predecessor will be analyzed using a merged set, the information about the source paths of these merged SOUS are lost. But, merging information does not make the analyzer give incorrect results as the integrity of the individual unsafe sets is preserved.

5 Handling Complex Programming Constructs

5.1 Handling Loops

A loop is detected when the predecessor of the current block is a block that has already been encountered in the analysis. Loops, if not properly handled, will result in the formation of wrong unsafe sets and incorrect results. So, as soon as the analyzer detects a loop, a new set called a *loop-set*, which is a set of set of unsafe sets, is created. That is, each element of the loop-set is a set of unsafe sets (thus, a loop-set is a set of SOUS). The analyzer also remembers the starting and the ending points (blocks) of the loop.

The first element added to the loop-set is the SOUS that the analyzer has computed when it detects the cycle. The analyzer then uses the current SOUS to analyze all the blocks forming the cycle including all the possible paths involving those blocks in the control flow graph. When the end block for the loop is reached, the analyzer checks if the current SOUS is already a member of the loop-set.

If the current SOUS is already a member, then the analyzer has reached a *fixed point* for the loop. That is, the analyzer has collected all the information from the loop and additional cycles through the loop will not add any new information to the sets. At this point the analyzer can safely exit the loop and continue the analysis of other unvisited blocks.

If the current SOUS is not already a member of the loop-set, then the analyzer merges (through the $\sqcup$ operation in the lattice) the current SOUS and loop-set and continues to cycle through the loop with the current SOUS, until a fixed-point is reached.

Loops can be nested and in that case, the above procedure is performed for the each of the inner loops. The fixed point is re-calculated for each of the inner loops for each cycle through the outer loop until the outer loop reaches a fixed point.

5.2 Handling Arrays

Operations on arrays, e.g.,

```
...
int a[] = {...};
...
a[..] = ....;
...
```

exactly resemble pointer operations at the assembly level. That is, if statically allocated array elements are accessed, then the corresponding assembly code will resemble the dereferencing of pointer variables. The analyzer handles these cases by looking at what the destination location of the access corresponds to, i.e., whether it is in the stack or in the heap.

5.3 Pointers and Global Variables

There are two cases in which global variables influence the analysis. The first case arises when a pointer (local variable) is assigned a value from (say an integer after typecasting) a variable which is declared as global. The second case arises when a pointer which is declared as a global variable is dereferenced. In both cases interprocedural analysis is needed.

Consider that we have a pointer `p` dereferenced in a function and assigned a global variable. Since the variable is global, it could be modified by any part of the program. If there is no assignment to that global variable in the current function prior to dereferencing, then its value is coming from outside the function, and a global interprocedural analysis has to be performed. Similarly when a pointer, declared as global, is dereferenced, a similar situation arises. Interprocedural analysis is addressed in the next section.

5.4 Handling Functions

As already discussed, the analyzer first splits any given program into functions and analyzes each function for safety. However, inter-procedural analysis is needed to detect hard-coding in the following cases: (i) return values of functions may be hard coded pointers that are later dereferenced in the calling function; (ii) arguments can be passed as a reference to functions and the called function hard codes the arguments and the calling function uses the hard coded values; and (iii) function bodies have statements involving either a globally declared pointer or a pointer that is assigned an expression involving global variables.

We compute and keep track of the calling and return context for each function in a memo table. Thus, for each function, values are set in the memo table to denote whether the function arguments or return values are hard-coded. This memo table is built iteratively (since functions may be mutually recursive) until it reaches a fix point, during the analysis of the program. Similar iterative inter-procedural analysis is used in analyzing hard codedness of global pointers and pointers whose value depends on global variables.

Our analyzer pre-processes the flow graph to determine if some of the functions can be analyzed "stand alone." These are analyzed and their resulting contexts stored in the memo-table in advance. Other functions are iteratively analyzed as outlined above.

5.5 Pointers requiring multilevel dereferencing

The usage of double pointers `(**)` or multilevel pointers that are hard coded complicates the analysis. This is because double pointers can be used to indirectly hard-code other single level pointer variables.

When single level pointers (say int *p) is used for hard coding, the typical sequence of operations for hard-coding are as shown below:

```
... somefunction(...)
```

```
{
declare pointer variables;
hard code the pointer variables;
create aliasing between the pointers;
dereference the pointers;
}
```

Aliasing two pointers before hard coding one of them, will not affect the analysis, since after hard coding, the two pointers will no longer be aliased. However, in the case of doubly (or more) indirected pointers there can be sequences such as the following.

```
... somefunction(...)
{
declare pointer variables;
create aliasing between the pointers;
hard code the pointer variables;
dereference the pointers;
}
```

A concrete example is shown below:

```
void main()
{
int *p, val;
int **q = &p;
//p is hard-coded via q
*q = (int*)0x8000;
val = *p;
}
```

Detection of hard-coding in these indirect cases involves analyzing each line of the code multiple times or carrying huge sets of aliases, making the analysis very costly.

The analyzer that we have built flags a warning when it sees multilevel pointer dereferencing. Note that the analyzer will detect hard-coding for double pointers that are hard-coded directly, i.e., occurrences such as `(int **)p=90; **p;` will be detected without any extra effort. Multilevel pointers cause problems in analysis only if we have these pointers used in hard-coding other pointers indirectly.

5.6 Handling Parallel Instructions

The `||` characters in the disassembled code signify that an instruction is to execute in parallel with the previous instruction[12].

```
   instruction A
|| instruction B
|| instruction C
```

For example, if a code sequence as shown is encountered, where instructions A, B, C are some assembly level instructions then it means that instructions A, B and C are executed in parallel. That is, the instructions A, B and C in the fetch packet correspond to the same execute packet and are executed in the same cycle. Moreover, instructions A, B and C do not use any of the same functional units, cross paths, or other data path resources.

Static analysis of disassembled code needs to make sure that it handles such kind of parallelism. Our analyzer does take care of such parallelism. As soon as dereferencing of a base register or occurrence of an element in the unsafe set (as the destination register) is found to occur in parallel with other instructions, the analyzer continues analysis with the instructions that occur in the previous cycle for that register or matched element.

6 Illustrative Examples

In this section, we include some illustrative examples to show the capabilities of the analyzer developed.

```
Example 1:
void main()
{
 p = ...;
 q = 0;
 for(i=0;i<p;i++)
 q++;
 *q;
}
```

In example 1, `q` is a NULL pointer that is dereferenced after being modified in the ‘for’ loop. The analysis is able to detect that `q` is hard coded. Note that the analysis would have detected `q` to have dereferenced a null pointer if `q` had not been updated in the ‘for’ loop.

```
Example 2
void main()
{
 int *p, *q, i;
 q = malloc(sizeof(int));
 i = (int) q;
 p = (int*) i;
 *p;
}
```

In example 2, though `p` is assigned an *int* value, that value was derived from a safe pointer. Our analyzer will also detect `p` to be safe.

```
Example 3
Void main()
{
 p = hard coded;
 q = good pointer;
 r = q + p - q;
 *r;
}
```

In example 3, though the dereferenced pointer `r` is derived as a function of a good pointer and a hard coded pointer, in reality we are assigning to `r` the hard coded pointer `p`. But since all pointer operations are abstracted, our static analyzer will not be able to detect this hard coding of pointer `r`.

7 Performance Results

	t_read	timer1	mcbsp1	figset	m_hdrv	dat	gui_codec	codec	stress	demo
Num fns	1	2	1	2	6	5	8	7	16	16
Num lines	80	126	196	292	345	950	1139	1188	1202	1350
Num BB	7	2	15	30	22	72	48	49	28	78
Num *ptr	3	17	0	19	6	10	102	109	105	82
Num HC	0	6	0	10	2	8	40	28	0	47
Max US size	0	1	0	2	1	3	1	1	1	2
Avg US size	0	1	0	2	1	2	1	1	1	1
Max SOUS size	0	13	0	8	1	5	9	4	4	13
Avg SOUS size	0	13	0	4	1	2	3	3	4	5
Max Chain Len	0	1	0	2	1	12	1	1	1	9
Avg Chain Len	0	1	0	2	1	3	1	1	1	2
RT (ms)	1280	1441	1270	1521	2262	2512	3063	3043	4505	4716

Table 2: Performance Results for the analyzer

The performance figures for our analyzer are given in Table 2. The first column corresponds to the metrics that are used to guage the performance of the analyzer, the subsequent columns show the performance for each of the selected DSP program.

The programs that were used to test were taken randomly from the TI's distribution of CCS[10]. We took the a.out files of the randomly selected programs, disassembled it, and then fed the assembly code to the analyzer. The analyzer produced a log file that contained the results of analysis and Table 2 was generated from the log file.

The value 'Num fns' corresponds to the number of functions in the program, 'Num lines' is the number of lines in the input file subject to analysis, while 'Num BB' is the number of basic blocks in the input file. 'Num *Ptr' is the number of pointers that were dereferenced in the file. 'Num HC' is number of hard coded occurrences in the program detected by the analyzer. 'Max US size' and 'Avg US size' are the maximum number and the average number of elements respectively in the unsafe sets created by the analyzer. 'Max SOUS size' and 'Avg SOUS size' are the maximum and average number of elements in the set of unsafe sets respectively. 'Max Chain Len' is the max path length from the point at which dereferencing occurs to the point at which the analyzer detects that the pointer is assigned a value. 'RT (ms)' is the running time of our analyzer in milliseconds.

From Table 2, we find that the average number of elements in the unsafe set at any point of time is small. One of the reasons is that modifications to the unsafe set always involve the deletion followed by the addition of one or more elements that got related to the deleted element. This also means that the number of elements to which a pointer gets related to through pointer arithmetic, assignment and other pointer operations is always small. Also, note that the number of element in the SOUS was a maximum of 13. This means that the maximum number of pointers that the analyzer was analyzing simultaneously is 13. The maximum chain length was 12 and in most cases the chain length is either 1 or 2. This means that most pointers were dereferenced immediately after getting hard coded. The amount of time that the analyzer takes to run depends on the number of hard coded pointers, the number of basic blocks in the disassembled code and the number of lines of code in the input file. The maximum time that the analyzer spent on analysis was less than 5 seconds on code with 1350 lines.

The results were produced by the prototype implementation of the analyzer without any fine tuning. The main aim of building the current prototype was to prove the efficacy and viability of static analysis based tools to perform these checks in commercial software when source code is not available. While the largest program we have tried was only 1350 lines long, the sizes of the unsafe sets are relatively small causing the fix-point computation to converge rapidly. Thus, we are reasonably confident that our analyzer will scale up for larger code sizes (work is in progress to obtain larger benchmarks to test our system). Also, it should be noted that a component binary needs to be statically analyzed only once before being integrated into an application.

8 Related Work

There is a wealth of literature on static program analysis. These static analyses either analyze data-flow or control-flow [13] of the program or employ an abstract interpretation based framework

[14, 15]. However, much of this work has been done in a scenario where source code is available. Not as much attention has been levied on analyzing machine code. Several researchers have looked at *link time optimization* [16, 17, 18, 19, 20, 21], where machine code has to be analyzed, however, as pointed out by Debray et al [8], they are "limited to fairly simple local analysis." There is a wealth of literature on pointer analysis [1, 2, 3, 22, 23, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 8, 5, 6, 7]. However, only a limited number of these [8, 5, 6, 7] consider analyzing machine code statically. Most of these efforts are concentrated on doing aliasing analysis of machine code, i.e., detecting whether two instructions will access the same memory location or not. Many of the issues that arise are similar to the ones that arise in our analysis, since these analyses have to keep track of *use-def* chains as well (i.e., given a use of a register, check where it was modified).

Debray et al [8] use a *mod k* abstraction in which no distinction is made between two addresses that have the same lower k bits. An interprocedural, context-sensitive data flow analysis is performed to see if two instructions access the same abstract address. Fernandez and Espasa [6] extend this analysis a little further by also analyzing if a memory reference is to the heap, stack or global memory. Amme at al [5] perform a similar analysis to aid parallelizing compilers. They symbolically abstract the values of registers which are then propagated in the flow graph; dependence information is then gathered via data-flow analysis. Analysis of assembly code via data-flow analysis has also been used for other applications, these include estimating memory use and execution time for interrupt driven software [4] and verifying security properties [7].

Static analysis has been recognized as an important technology for software quality assurance [48, 46], however, the limited efforts described in the literature primarily analyze the source code [47, 45, 46, 53, 48, 52]; none of them deal with standard compliance. Those that analyze assembly code are only interested in security properties [7, 42, 43, 44] and not in software programming standard compliance. Thus, to the best of our knowledge there is no existing work that statically analyzes embedded assembly code to check for software standard compliance.

Other related work includes work done by software groups at Texas Instruments to develop tools for automatic compliance check. Static analysis based approach was considered too costly for the benefits obtained, and instead a testing-based approach was resorted to where a program code is run in different scenarios (for example, different parts of the memory) and checked to see if erroneous output is obtained. Of course, this method is not complete either. It also cannot pinpoint the problem (for example, the pointer that is hard coded cannot be automatically identified; all we can conclude that the code has *some* compliance problem). Our results in this papers show, that contrary to belief, a static analysis based technique is effective, practical, and useful.

9 Conclusions and Future Work

In this paper, we presented an abstract interpretation based static analysis framework for analyzing hard-codedness of pointer variables in embedded assembly code. Our results show that static analysis based approaches are viable in industrial settings for checking for coding standards compliance. Code compliance checking is critical for code reuse and COTS compatibility in applications. A complete analyzer has been developed for pointer hard-codedness analysis and shown to run successfully on code samples taken from Texas Instruments' DSP code suite. The prototype system is currently being refined to provide more accurate results in presence of global pointers and mutually recursive functions. Future work also includes extending the system to handle rules 1, 2, and 4 through 6 [11] laid out by TI. Note that the analyses needed for rule numbers 4 and 6 are very similar to hard-codedness analysis. Similarly rules 2 and 5 require analysis that determines if a binary code is re-entrant. Note that the analysis for determining if the code is re-entrant is similar to pointer hard codedness analysis. A binary is re-entrant if it does not contain any instructions that will modify a location in the code area (i.e., where the binary resides in the main memory). Thus, once again all memory locations can be divided into two sets: those that correspond to the code area and those that do not. Each instruction then has to be analyzed to ensure that if it performs a memory-write operation, then the target location is in the latter set. Once again, similar to the hard-codedness analysis, we can abstract the two sets by constants CA (for code area) and NCA (non-code area). A lattice can then be built just as was done for the sets corresponding to HC and NHC, and an abstract interpretation based analysis performed in a manner described in 3. Work is in progress in this direction.

Acknowledgments

We are grateful to Steve Blonstein of Texas Instruments XDAIS group for discussions.

References

[1] Samuel Z. Guyer, Calvin Lin. Client-Driven Pointer Analysis. Static Analysis Symposium. 2003. Springer LNCS 2694. pp. 214-236.

[2] S. Adams, T. Ball, et al. Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis. SAS 2002. pp. 230-246.

[3] Donglin Liang, Mary Jean Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. SAS 2001. Springer LNCS 2126. pp. 279-29.

[4] D. Brylow, N. Damgaard, J. Palsberg, Static Checking of Interrupt-driven Software. International Conference on Software Engineering. 2001.

[5] W. Amme, P. Braun, E. Zehendner, F. Thomasset. Data Dependence Analysis of Assembly Code. Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98).

[6] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. International Conference on Parallel Architectures and Compilation Techniques. 2002.

[7] J. Bergeron, M. Debbabi, M.M. Erhioui, B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999. Palo Alto, California

[8] Saumya Debray, Robert Muth, Matthew Weippert Alias analysis of executable code. POPL'98.

[9] Texas Instruments Code Composer Studio Getting Started Guide, Literature No: SPRU509C.

[10] Texas Instruments TMS320C6000 Code Composer Studio Tutorial, Literature No: SPRU301C.

[11] TI TMS320 DSP Algorithm Standard Rules and Guidelines, Literature No: SPRU352D.

[12] TI TMS320C6000 CPU and Instruction Set Reference Guide, Literature No: SPRU189F.

[13] Alfred V.Aho, Ravi Sethi and Jeffrey D.Ullman Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1988.

[14] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. Fourth Annual ACM Symp. on Principles of Programming Languages. 1977. pp. 238-252.

[15] S. Abramsky and C. Hankin Abstract Interpretation of Declarative Languages, Ellis Horwood, 1987.

[16] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, Spike: An Optimizer for Alpha/NT Executables, Proc. USENIX Windows NT Workshop, Aug. 1997.

[17] D. W. Goodwin. Interprocedural Dataflow Analysis in an Executable Optimizer. Proc. PLDI '97. pp 122-133.

[18] E. Ruf, ContextInsensitive Alias Analysis Reconsidered. Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation. June 1995, pp. 13–22.

[19] A. Srivastava and D. W. Wall, Linktime Optimization of Address Calculation on a 64bit Architecture. Proc. PLDI 1994. pp. 49–60.

[20] B. Steensgaard. Pointsto Analysis in Almost Linear Time, Proc. 23th. ACM POPL 1996, pp. 32–41

[21] W. E. Weihl, Interprocedural data flow analysis in the presence of pointers, procedure variables, and label vari ables. Proc. ACM POPL. 83–94.

[22] D. R. Chase, M. Wegman, and F. K. Zadeck, Analysis of Pointers and Structures. PLDI '90. June 1990, pp. 296–310.

[23] J.D. Choi, M. Burke, and P. Carini, Efficient FlowSensitive Interprocedural Computation of PointerInduced Aliases and Side Effects. Proc. 20th ACM POPL. 1993. pp. 232–245.

[24] K. D. Cooper and K. Kennedy, Fast Interprocedural Alias Analysis. 16th ACM POPL. Jan. 1989, pp. 49–59.

[25] D. Coutant, Retargetable HighLevel Alias Analysis. Proc. 13th ACM POPL. 1986, pp. 110–118.

[26] A. Deutsch, On determining lifetime and alias ing of dynamically allocated data in higherorder functional specifications. Proc. 17th ACM POPL. 1990, pp. 157–168.

[27] A. Deutsch, Interprocedural MayAlias Analysis for Pointers: Beyond klimiting. PLDI 1994. pp. 230–241.

[28] A. Diwan, K. S. McKinley and J. E. B. Moss, TypeBased Alias Analysis. Manuscript, Dept. of Computer Science, UMass. Amherst, 1996.

[29] M. Emami, R. Ghiya and L. J. Hendren, ContextSensitive Interprocedural Pointsto Analysis in the Presence of Function Pointers. Proc. SIGPLAN PLDI '94. pp. 242–256.

[30] S. Horwitz, P. Pfeiffer, and T. Reps, Dependence Analysis for Pointer Variables. Proc. PLDI '89. pp. 28–40.

[31] J. Hummel, L. J. Hendren, and A. Nicolau, A General Data Dependence Test for Dynamic, PointerBased Data Structures", Proc. PLDI '94. pp. 218–229.

[32] N. D. Jones and S. S. Muchnick, Flow analysis and optimization of LISPlike structures. In *Program Flow Analysis*. Prentice Hall, 1981, pp. 102–131.

[33] W. Landi and B. G. Ryder, Pointerinduced Aliasing: A Problem Classification. Proc. 18th ACM POPL. 1991, pp. 93–103.

[34] W. Landi and B. G. Ryder, A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. Proc. SIGPLAN PLDI '92. pp. 235–248.

[35] J. R. Larus and P. N. Hilfinger, Detecting Conflicts Between Structure Accesses. PLDI '88. pp. 21-34.

[36] E. Ruf, ContextInsensitive Alias Analysis Re considered, Proc. PLDI '95. June 1995, pp. 13–22.

[37] M. Shapiro and S. Horwitz,Fast and Accurate FlowInsensitive PointsTo Analysis, Proc. ACM POPL. pp. 1–14.

[38] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. Proc. ACM POPL. Jan. 1980. pp. 83–94.

[39] R. P. Wilson and M. S. Lam. Efficient Context Sensitive Pointer Analysis for C Programs. Proc. PLDI '95. pp. 1–12.

[40] M. Wolfe, Optimizing Supercompilers for Super computers, MIT Press, Cambridge, Mass., 1989.

[41] H. Zima and B. Chapman, Supercompilers for Parallel and Vector Computers. ACM Press. New York. 1991.

[42] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. 12th USENIX Security Symposium, August 2003.

[43] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie and N. Tawbi. Static Detection of Malicious Code in Executable Programs. Symposium on Requirements Engineering for Information Security (SREIS'01).

[44] J. Bergeron, M. Debbabi, M. M. Erhioui and B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. In Proceedings of the IEEE 4th International Workshops on Enterprise Security (WETICE'99).

[45] B.V. Chess. Improving computer security using extending static checking. IEEE Symposium on Security and Privacy, 2002.

[46] David A. Wagner. Static analysis and computer security: New techniques for Software Assurance. University of California at Berkley Phd Dissertation. Dec. 2000.

[47] Hao Chen, Jonathan S. Shapiro. Exploring Static Checking for Software Assurance. SRL Technical Report SRL-2003-06.

[48] Improving software quality by static program analysis Horst Licheter and Gerhard Riedinger Proc. of SPI 97 software process improvement, Barcelona, 1997

[49] Demand-Driven Pointer Analysis Nevin Heintze, Oiivier Tardieu Conference on Programming Language Design and Implementation 2001

[50] Points-to Analysis in Almost Linear Time (1996) Bjarne Steensgaard Symposium on Principles of Programming Languages

[51] M. R. Garey and D. S. Johnson. Computers and Intractability. W. H. Freeman and Company. New York. 1979.

[52] George C. Necula Scott McPeak Westley Weimer CCured Type-Safe Retrofitting of Legacy Code POPL '02, Jan. 16-18, 2002 Portland, OR USA

[53] Gerard J. Holzmann. Static Source Code Checking for User-defined Properties. Conference on Integrated Design and Process Technology, IDPT-2002.