


AHMED M. BAKR 
MAY SALAMA
ABDELWAHAB K. ALSAMMAK

HIERARCHICAL STATE MACHINE MODEL FOR ANALYZING SAFETY HAZARDS IN REAL-TIME SYSTEMS

Abstract *Real-time systems must avoid hazardous situations. To achieve this, their functionality should be investigated under time constraints. A model based on Hierarchical Communicating Real-time State Machine (H-CRSM) and analysis methodology is proposed in this paper with the objective of obtaining any hazardous events that may occur in the input ANSI-C program. The system outputs a scenario list of the different hazards. A path in the code showing the cause of the undesirable event is associated with each hazardous scenario. The strength of the proposed methodology is that the process of hazardous situation detection does not require the running of the ANSI-C program many times with distinct values for the inputs. It also focuses on analyzing the software level of the life cycle. It is not like most of the verification and analysis tools that check system levels. The system level may be bug-free, but the software level may not be.*

Keywords fault tree analysis, hazard analysis, static code analysis, CLANG, HCRSM, ANSI-C, safety critical real-time systems

Citation Computer Science 22(1) 2021: 39–80

Copyright © 2021 Author(s). This is an open access publication, which can be used, distributed and reproduced in any medium according to the Creative Commons CC-BY 4.0 License.

1. Introduction

Static analysis is the process of finding bugs automatically by using algorithms and techniques that analyze the source code. This operation is indeed slower than the compilation process, as it requires some of the algorithms to run in exponential time. A static analysis can detect bugs falsely in a code, whereas the code correctly functions. The false positive frequency in different code checkers vary remarkably due to the different analysis precision of each checker.

With today's rapid increase of our dependence on electronic complex devices, a very important attribute has emerged, which is the functional safety of the device. Functional safety testing [24] is not only about producing the correct output for specific inputs. It is most importantly about producing the correct output in the right time. For some systems, real-time testing would be very difficult as well as costly and might not cover all of the likely hazardous situations [6, 21, 28].

Accidents that have recently happened due to software problems are the trigger of our proposed work. These accidents have drawn our focus to find a technique to detect and locate hazards in software. In the air navigation field, the unfortunate crash of the Ethiopian Airlines jet in March 2019 [11] was one of the worst airplane accidents. Another crash of the same airplane model preceded this one by five months [23]. Casualties were reported in both crashes. The preliminary report stated a switch malfunction [11]. In the automotive field, a Tesla Model 3 drove under the trailer of a truck in March 2019 [31] when the autopilot mode was working. This accident is very similar to the one that happened in 2016 due to the autopilot mode. Again, casualties were reported. Toyota recalled many cars in 2010 because of a bug in the anti-lock braking software [32].

Other domains like medical devices or military equipment fall into the critical domains [20] that directly affect the safety of individuals. Since C language is looked upon as one of the fastest high-level languages that deal with no limitations with hardware, most of the real-time systems are American National Standards Institute for C programming language (ANSI-C) implemented.

In this study, a methodology for analyzing the hazardous events that could happen in safety-critical systems is proposed. The block diagram of the proposed method is depicted in Figure 1. The proposed system is composed of two sub-systems: a modeler (parser), and an analyzer. There are two inputs to the system: the ANSI-C project, and the hazardous event equation. The ANSI-C project, which is the first input, is parsed to produce a timed model where each C statement preserves its time attribute. The parsing process is based on the Hierarchical Communicating Real-time State Machine (H-CRSM) formal modeling strategy [12, 13, 29]. The hazardous event equation is the second input. The objective of this study is to automatically analyze the hazardous event equation to produce a hazardous scenarios list using the H-CRSM-generated model. Figure 2 describes all of the stages for the proposed system. The bright clouds are the system inputs. The dark clouds are the system outputs. The ovals are the stages of the system.

The proposed system proves that the hazard equation can be valid if it finds paths that can lead to the hazard equation being true. In this case, the root cause events are displayed to the user as proof that the hazard equation can be achieved.

This paper is constructed in six sections. Section 2 briefly introduces related works. Section 3 demonstrates the proposed modeler. Section 4 presents the proposed analyzer. Along the paper, a case study is explained to show how the diagram and H-CRSM model are generated. It is also used by the analyzer to show a real-world example on each of the Undesirable Event (UE)-elements in the analysis phase. Section 5 describes two case studies to have a full picture of the system that shows how the system can detect and locate hazards that could occur in the ANSI-C code. Finally, the conclusion of the work is in Section 6. The proposed system can be extended to any other programming languages. It is not limited to only C-codes.

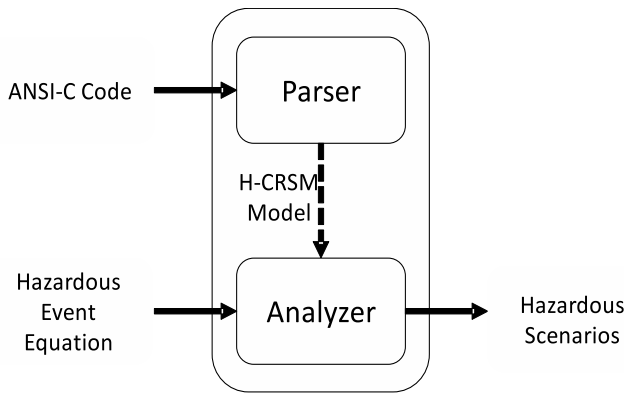


Figure 1. Block diagram of proposed system [4]

2. State of the art

Due to the importance of safety critical systems analysis, many researches have been carried out in this scope. In [27], the authors developed an approach called Hazard Analysis and Operability Analysis (HAZOP). This approach is used for generating test models using a safety analysis technique by generating timed automata from extended system requirements, which helps in building test cases. They added more alternative scenarios to the requirements by deriving guide phrases from the original requirements, which are used to generate alternative requirements scenarios.

In [14], an integration between the behavioral and fault models is done to generate more test cases for the system model. An aerospace application was used to validate their work.

Another approach is introduced in [25]. This is a combination of the automotive Hazard Analysis and Risk Assessment (HARA) and Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE) strategies. It defines the influence of security concerns on safety conceptions at the

system level. The approach classifies the security hazard probabilities; it is then used to determine the number of corrective actions to be taken. In his book [10], Ericson demonstrates the way to work out the mostly used hazard analysis techniques like Environmental Hazard Analysis (EHA), Fault Tree Analysis (FTA), failure mode and effects analysis, HAZOP, and Event Tree Analysis (ETA). In [22], a formal modeling and safety analysis framework is built for safety critical systems called S-sharp. This provides a domain-specific modeling language. It is an automated formal safety analysis tool that is built using the .NET and C-sharp languages.

In [19], the author demonstrates the incorporations between use cases and a modified method for functional hazard assessment. This depends on an early analysis of the hazard of the Unified Modeling Language (UML) at a functional level. The FTA method is presented in [9]. This decomposes system-level failures using tree structures. The decompositions result in lower-level events and logic gates that model their interactions. Another approach that employs Binary Decision Diagram (BDD) is presented in [2], where it attempts to solve the dependencies between the branch point events in FTA that produce incorrect analysis outputs. In [18], the authors developed a stochastic FTA model to calculate the probability of occurrence for a top-level event. In [17], Ishimatsu et al. proved that a traditional hazard analysis scope is the failure of components; nevertheless, the software continues to operate. Software can frequently be a factor in causing accidents. A method is demonstrated in [30] that repeatedly conducts Failure Mode and Effects Analysis (FMEA) and FTA until there are zero risks in the control software; thus, it makes the control software safer. FMEA is employed in order to analyze any control software risks. In [33], the authors focused on the formal modeling and verification of the Reactor Protection System (RPS) system in a nuclear power plant based on their long experience in the field. They used FTA templates to formalize the process of the requirements analysis. The Software Cost Reduction for Nuclear Applications (NuSCR) formal specification was used in the development process of the requirements analysis. The Computation Tree Logic (CTL) model checker was applied to verify and validate the requirements analysis. They built a system that automatically transforms the NuSCR specifications into a Function Block Diagram (FBD) program in the design phase. They used FTA templates for the design phase. Meanwhile, the FBD model and Verification Interacting with Synthesis (VIS) analyzer was applied to verify the model in the design phase. In [34], a theoretical approach based on System-Theoretic Process Analysis (STPA) Based on Hazardous Control Action Tree (HCAT-STPA) is developed to produce a model and analyze the hazards that may occur at the system level. This strategy is established on the System-Theoretic Accident Modeling and Processes (STAMP) and STPA analysis methods. The problem with STPA is that it depends excessively on individual inspection.

The Axivion Bauhaus Suite is a non-free tool for static code analysis for many languages, including C. It performs architecture checking, interface analysis, Motor Industry Software Reliability Association (MISRA) C checks, and clone detection [19].

Astrée is a non-free tool for static code analysis that aims to prove that there are no runtime faults. It analyzes C codes that uses memory in a complex way; there is neither dynamic memory allocation nor recursion. It targets embedded systems, nuclear energy, medical instruments, and aerospace applications [3]. Its objective is to make sure that the code satisfies the behavioral properties of the employed interface that it uses [7]. It is based on CPAchecker, which is a tool for configurable software verification [1]. Coverity is one of the leading static analysis tools in the automotive industry founded in the Computer Systems Laboratory at Stanford University [8]. It supports many platforms, including C-language. It analyzes more than 3,900 open-source projects. Infer is a free static-analysis tool developed by the Facebook team with open-source contribution that focuses on null-pointer, other memory problems, coding conventions, and unavailable APIs.

Infer [16] applies a technique called bi-abduction to analyze a program compositionally. This analysis interprets code procedures separately from their callers. It is declared that this helps the scalability of Infer to cope with big codebases as well as run fast on code changes in a step-by-step manner. Polyspace is one of the strongest tools used in the automotive industry; it is provided by Mathworks. It employs abstract interpretation to find out whether there are errors during runtime. Dead Code (also in the source code) is used to check all MISRA rules. It has two separate versions. The first is a polyspace code prover that proves that there are no critical run-time error without code execution. The second is Polyspace Bug Finder, which acts as a checker of coding rules, code metrics, and security standards; it also finds faults. The real problem with Polyspace Bug Finder is that it has too many false positives, which generates thousands of possible threads that are not actually valid threads [26]. Helix QAC is a strong non-free tool used in the automotive industry. It parses the C-code against all MISRA checks and raises errors if any of the rules are violated. It also has the ability to prioritize errors based on their severity [15].

From the brief survey above, it is noticed that the focus of the above-mentioned work is hazard analysis during the modeling phase. However, hazards are susceptible to occurrence during the implementation phase; yet, no hazards are detected in the model. Industry static analysis tools try to find hazards from the implementation phase of the system. However, most of them focus on running MISRA checks against the code to find those rules that the input code violates. Very few tools try to find hazards based on well-known patterns. This strategy results in thousands of false positive errors. Most of these errors are neglected by the user of the tools.

3. The proposed modeler

The ANSI-C source code is reverse-engineered into a model that can be analyzed. An introduction to the modeling system (see Fig. 2) can be found in our paper [5]. The input C-project is pre-compiled using Gnu's Not Unix (GNU) Compiler Collection (GCC) to solve all includes, definitions, macros, and processor-predefined commands. The pre-compiled project is fed into the model system to generate the real-time model.

The model is fed into the analyzer with the hazard equation. The hazard equation is the condition that we do not want to happen in our system. The analyzer output is a highlighted list of conditions in the code. Each output condition shows a path that causes a hazard equation to occur. The hazard scenarios are passed to the table-generation module. The variables are extracted from each hazard scenario and the user is asked to define the valid classes for each variable. The table-generation module generates a list of valid values for each variable to satisfy that hazard scenario. The generated list of values for a hazard equation is the proof that the input system is vulnerable.

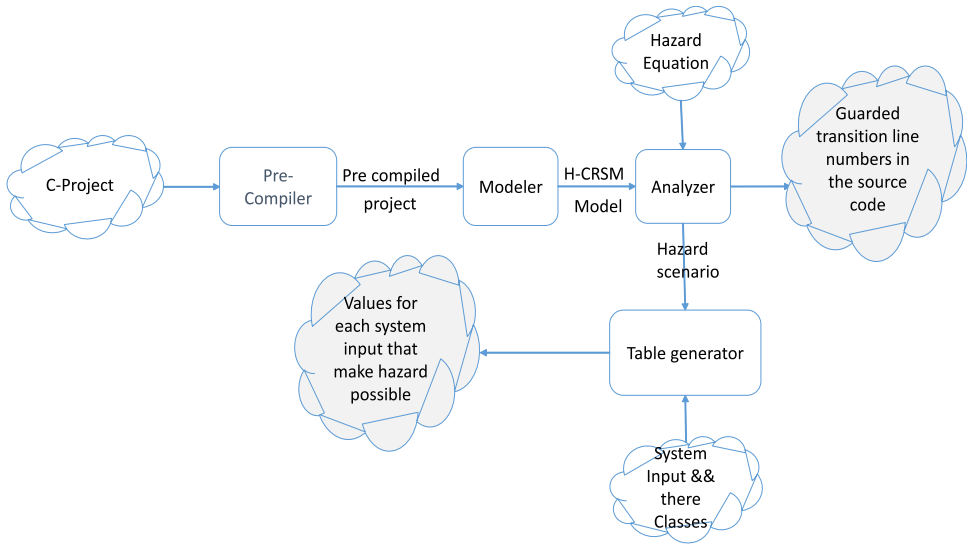


Figure 2. Proposed system hierarchy

The modeling system consists of three phases as shown in Figure 3 [5]. The Abstract Syntax Tree (AST) is generated when the precompiled ANSI-C code is parsed and the H-CRSM model [12, 13] is generated when the AST is traversed.



Figure 3. Modeler block diagram

A comparison table in our modeling paper [5] summarizes the best-known compilers. It was proven that the best option is the C Language front-end compiler (C-Lang) parser because it generates AST, it is implemented in C++ (which is very fast in processing), and uses the American National Standards Institute (ANSI) C99 and ANSI C11 standards. The model is represented by H-CRSM due to its capability to represent time in the model. It is derived from the Communicating Real-time State

Machine (CRSM) model [29]. The hierarchical modules added to H-CRSM helped in making the system expandable.

The AST generated from the C-Lang parser is traversed, then the H-CRSM model is generated. The model contains one input C-project as shown in Figure 4 [5]. Each project contains one extern global machine that has the definition for all extern variables defined in the project. It also has one or more C-files. Each C-file has only one global machine that contains the definition for all of the global variables defined in this file. It also has one or more machine(s) that is/are the representation for a C-function in the model. Each machine contains one function-container, a list of function parameters, a list of acceptance states that corresponds to the return statements in C-language, a list of machines that called this machine, and a list of machines that this machine calls. Each container has zero or more variable definitions, zero or more children containers, and zero or more children transitions. A container is a block in C-language. A block can be an if-container, switch-container, loop-container, function-container, or self-compound-container.

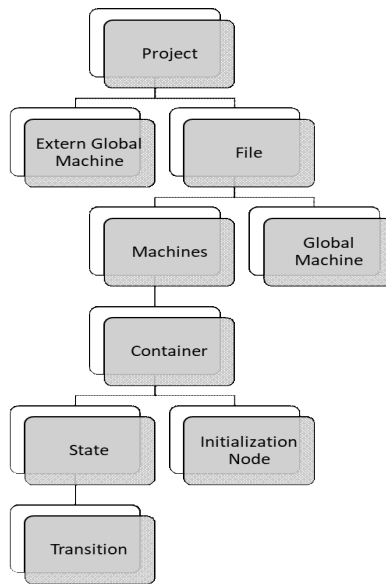


Figure 4. Modeler hierarchy diagram

Figure 5 [5] shows a UML diagram for the container class. An average execution time is added for each transition to define the time needed to execute the command on that transition. A guard condition may be added to the transition to block the execution of the transition if its guard condition is not satisfied. It also has a source state and a destination state. Each state has a list of input transitions and a list of output transitions. A guarded-transition is a transition that has a guard-on has one

atomic C-statement. When a transition is executed, its corresponding C-statement is executed as well.

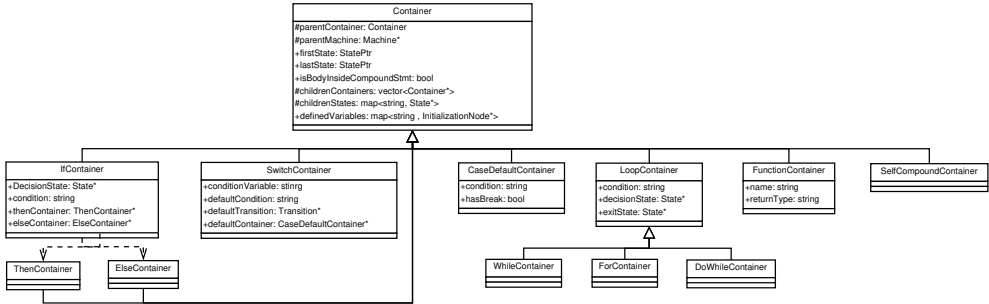


Figure 5. Container class diagram [5]

This section discusses the translation of some elements in C-language into the H-CRSM model. Section 3.1 describes the function representation in the H-CRSM model. Section 3.2 describes the variable representation in the H-CRSM model. Section 3.3 describes the array representation in the H-CRSM model. Section 3.4 describes the pointer representation in the H-CRSM model. Section 3.5 describes the enumeration representation in the H-CRSM model. Section 3.6 shows a C-code example and its translation to an H-CRSM example in a state machine diagram and XML formats.

3.1. Function declaration

Function declaration algorithms are executed when entering a new function or exiting from a function after finishing its body.

The system creates a new machine as discussed in Algorithm 1. The newly created machine becomes the current machine. Each machine is one big function-container. The function container parameters shown in listing 5 are set. Each machine has one or more state(s), so the system creates a new state.

Algorithm 1 Visiting function declaration algorithm [5]

- 1: **procedure** VISITFUNCTIONDECL
 - 2: **if** function-declaration is not a prototype **then**
 - 3: *current-machine* ← **create a new machine**
 - 4: *function-container.parent-machine* ← *current-machine*
 - 5: *function-container.name* ← *function-name*
 - 6: *function-container.return-type* ← *function-return-type*
 - 7: *current-machine.current-container* ← *function-container*
 - 8: *current-machine.current-state* ← **create-new-state**
 - 9: *current-machine.beginning-state* ← *current-machine.current-state*
 - 10: *current-machine.current-transition* ← **NULL**
 - 11: *function-container.first-state* ← *current-machine.current-state*
-

Algorithm 2 is executed when exiting a function declaration, which means that all children within the function body have been visited. The system checks whether the current-state is a useless state, which means it has no input or output transitions and is not an acceptance state. The system then resolves all transitions whose *destination-state* is not set yet; then, it assigns a NULL value to the *current-state*.

Algorithm 2 Exiting function declaration algorithm

```

1: procedure EXITFUNCTIONDECL
2:   function-container.last-state ← current-machine.current-state
3:   if current-state is useless then
4:     delete current-state from current-machine.states-list
5:   resolve-unfinished-goto-transitions
6:   current-machine ← GLOBAL-MACHINE
  
```

3.2. Variable representation

Each variable is represented as a node that has a list of read-access instances that defines all transitions where this variable is read. It has a list of write-access instances that defines all transitions where this variable is written. It has also a list of pointers that point to this variable.

Algorithm 3 is executed when entering a new declaration statement in the parse tree. The system first checks whether this variable declaration statement is the first statement after a go-to statement like **goto label; int x;**, then, a new state must be created to be the source state of the variable declaration statement, as there is no relationship between the GOTO statement and the variable declaration statement.

Algorithm 3 Visiting new variable declaration algorithm

```

1: procedure VISITVARIABLEDECL
2:   add-new-state-if-first-stmt-after-goto-or-return-stmt
3:   if is-extern-global-variable then
4:     saved-machine ← current-machine
5:     current-machine ← GLOBAL-EXTERN-MACHINE
6:   add-var-init-node-to-init-machine(variable-declaration)
7:   if variable-declaration is an array then
8:     array-declaration(initialization-statement)
9:   else if is initialization exists and is ternary operation exists then
10:    visit-var-decl-when-init-part-is-ternary-stmt
11:   else
12:    variable-declaration(variable, initialization-statement)
13:   if it is a function-parameter variable-declaration then
14:    current-machine.add-function-parameter(variable-name)
15:   if is-extern-global-variable then
16:    current-machine ← saved-machine
  
```

If the variable is an extern global variable, then its definition must be put inside the extern global machine, so the algorithm sets the current machine as the extern global machine in Lines 3–5, and before the end of the function, the algorithm sets the current-machine back to the saved-machine in Lines 15–16.

The system adds a new variable initialization node to the initialization machine in Line 6. It checks that this variable is an array variable declaration; then, it calls an **array-declaration** procedure, which is described in Algorithm 10. If the variable is not an array variable declaration, then it checks whether a ternary initialization part exists; then, it calls **visit-var-decl-when-init-part-is-ternary-stmt** described in Algorithm 4. Finally, if no ternary operation exists, then the algorithm calls a **variable-declaration** procedure, which is discussed in Algorithm 6. Note that a variable declaration may be a scalar C-type, structure, union, or enumeration.

Algorithm 4 is responsible for handling the ternary initialization statement, as it treats the current-state as a conditional state that has two guarded input transitions; this means that the first transition will execute when the ternary condition is satisfied as shown in Algorithm 5 and the second guarded transition will execute when the ternary condition is not satisfied.

Algorithm 4 Visit variable declaration when initialization part is ternary statement

```

1: procedure VISITVARDECLWHENINITPARTISTERNARYSTMT
2:   conditional-state  $\leftarrow$  current-machine.current-state
3:   visit-var-decl-when-ternary-condition-is-satisfied
4:   condition-satisfied-last-state  $\leftarrow$  current-machine.current-state
5:   visit-var-decl-when-ternary-condition-is-not-satisfied
6:   condition-not-satisfied-last-state  $\leftarrow$  current-machine.current-state
7:   condition-satisfied-last-state.merge(condition-not-satisfied-last-state)

```

Algorithm 5 is executed when a ternary condition is satisfied. A transition is created from the *conditional-state* to the newly created *condition-satisfied-state* with guard condition *if(ternary-condition)*. When the ternary condition is not satisfied, the algorithm is very similar to the algorithm when the ternary condition is satisfied. The only difference is that the guard condition will be negated as follows *if(ternary-condition == false)*.

Algorithm 5 Visit variable declaration when Ternary condition is satisfied

```

1: procedure VISITVARDECLWHENTERNARYCONDITIONISSATISFIED
2:   condition-satisfied-state  $\leftarrow$  create-new-state
3:   current-transition  $\leftarrow$  create-new-transition(conditional-state, condition-satisfied-state,
   if(ternary-condition), NULL)
4:   give-read-access-to-stmt(ternary-statement.condition)
5:   variable-declaration(variable, initialization-statement)

```

Algorithm 6 is executed when defining a new variable. A variable's type may be a structure, union, enumeration, or c-scalar type. If the variable type is a structure, then Algorithm 7 is executed. If the variable type is a union, then an algorithm very similar to the structure algorithm is executed. If the variable type is an enumeration, then Algorithm 8 is executed. Finally, if the variable type is a C-scalar type, then Algorithm 9 is executed.

Algorithm 6 Variable declaration

```

1: procedure VARIABLEDECLARATION(variable, initialization-statement)
2:   if variable.type is a structure then
3:     structure-declaration(variable, initialization-statement)
4:   if variable.type is a union then
5:     union-declaration(variable, initialization-statement)
6:   if variable.type is an enumeration then
7:     enumeration-declaration(variable, initialization-statement)
8:   if variable.type is a C-scalar-type then
9:     c-scalar-type-declaration(variable, initialization-statement)

```

Algorithm 7 is responsible for defining a structure. It loops through all of the *structure-node* children, gets the initialization statement for this *structure-child* if it exists, and calls **variable-declaration**, which takes care of the recursive part.

Algorithm 7 Structure declaration

```

1: procedure STRUCTUREDECLARATION(structure-node, init-stmt)
2:   for each structure-child in structure-node do
3:     structure-child-init-stmt ← get-next-initialization-statement(init-stmt)
4:     variable-declaration(structure-child, structure-child-init-stmt)

```

Algorithm 8 Enumeration declaration

```

1: procedure ENUMERATIONDECLARATION(variable, init-stmt)
2:   create-var-access-data-if-doesnt-exist(enum.var-name, enum.type-name)
3:   if initialization-statement-exists then
4:     destination-state ← create-new-state
5:     current-transition ← create-new-transition(current-state, destination-state, NULL,
declaration-statement)
6:     get-variable-access-data-by-variable-name(enum.variable-name)
7:     create-variable-access-instance(write-access, current-transition)
8:     manually-traverse-init-stmt-and-give-them-read-access(init-stmt)

```

Algorithm 9 is responsible for declaring a scalar C-type variable like int, float, char, etc. First, the system creates a variable access data record for this variable (if it does not already have one). It checks whether the variable is inside a function, then it creates the variable record inside the current machine variable list; or else, this variable is a global variable, and it creates the variable record inside the global-machine variables-list.

Algorithm 9 C-scalar-type declaration

```

1: procedure CSCALARTYPEDECLARATION(var, init-stmt)
2:   create-var-access-data-if-doesnt-exist(var.var-name, var.type-name)
3:   if init-stmt-exists then
4:     destination-state ← create-new-state
5:     current-transition ← create-new-transition(current-state, destination-state, NULL,
declaration-statement)
6:     get-variable-access-data-by-variable-name(var.variable-name)
7:     create-variable-access-instance(write-access, current-transition)
8:     manually-traverse-init-stmt-and-give-them-read-access(init-stmt)

```

The system checks whether an initialization part in Line 2 is defined for the variable, then a new transition is created from the current state to the created new state; then, it gets the variable access instance for this variable in Line 6 and creates a write-access instance to the Left-hand Side (LHS) of the initialization statement in Line 7. Finally, it adds a read-access instance to all variables appearing on the Right-hand Side (RHS) of the initialization statement as shown in Line 8.

The system is capable of dealing with multiple variables that have the same name, but defined in different scopes, as shown in Listing 1 where variable-name i is defined in three different scopes. The system is capable of dealing with multiple variables that have the same name, but defined in different scopes, as in Listing 1 variable-name i is defined in three different scopes. The first definition is the global variable. The second definition is inside the *while-loop*. The third definition is inside the *for-loop*.

Listing 1. Multiple definitions for variable example

```

1 int i;
2 int main(){
3     int x;
4     while(x < 5){
5         int i = 10;
6         //do some stuff
7         x--;
8     }
9     for(float i = 0; i < 3.2; i+= 0.1){
10        //do some stuff
11    }
12 }
```

3.3. Array declaration

Algorithm 10 is responsible for declaring an array. The system calls **recursive-array-declaration** to get all of the elements of the array. For example, consider an array of two dimensions called **arr** whose size is $M \times N$. The first array element is **arr[0][0]**. The last array element is **arr[M-1][N-1]**. In this example, the call to function **recursive-array-declaration** will result in $M \times N$ array elements.

Algorithm 10 Array declaration

```

1: procedure ARRAYDECLARATION(array-declaration, init-stmt)
2:   array-elements  $\leftarrow$  recursive-array-declaration
3:   for each array-element in array-elements do
4:     array-element-init-stmt  $\leftarrow$  get-next-init-stmt(init-stmt)
5:     variable-declaration(array-element, array-element-init-stmt)
```

For each element in the array, the algorithm calls the **get-next-init-stmt** that is responsible for getting an initialization statement from the list. For example, if an array is defined as $int\ array[3][2] = \{\{1, 2\}, \{3, 4\}, \{5, 6\}\}$, then the first call to the function will get the value 1, the second call will get the value 2, and so on.

The system then calls the **variable-declaration** procedure for that array element as discussed in Algorithm 6.

3.4. Pointer representation

Pointers deal directly with memory in the C language. They are usually used with arrays and fixed memory addresses.

As an example, *struct MyStruct *ptr*; represents a pointer to an array. When incrementing pointer *ptr++*;, it does not increment by one but increments by the size of structure *MyStruct*.

In the case of pointer initialization in C statement *int *ptr=&var*;, a write access is added to pointer initialization node *ptr*. The system adds pointer *ptr* to the *pointer-list*. The *pointer-list* exists inside the initialization node for variable *var*. Variable *var* is added to *pointee-variables* in the pointer initialization node for *ptr*.

Algorithm 11 discusses the assignment of a pointer. *p = arr*; is an example for a pointer assignment to an array. The algorithm adds the first element of the array to the pointer variables list if the pointee is an array as shown in Lines 6–8. The pointer variable list contains a list of variables to which this pointer points. If the pointer points to another pointer, then the other pointer is added to the pointer list as shown in Lines 4–5. The pointer list is a list that contains all of the pointers to which this pointer points. All of the variables pointed to by a pointer in the pointer list are pointed to by this pointer as well. If the pointer points to a variable, then this variable is added to the pointer variable list as shown in Lines 9–10.

Algorithm 11 Pointer assignment algorithm

```

1: procedure POINTERASSIGNMENT
2:   destination-state  $\leftarrow$  create-new-state
3:   current-transition  $\leftarrow$  create-new-transition(current-state, destination-state, NULL, assign-
   statement)
4:   if RHS-of-assign-operation is pointer then
5:     pointer-to-pointers-list.add(RHS-of-assign-operation)
6:   else if RHS-of-assign-operation is array then
7:     first-elem-in-arr gets get-first-element-in-arr(RHS-of-assign-operation)
8:     pointer-to-vars-list.add(first-elem-in-arr)
9:   else
10:    pointer-to-vars-list.add(RHS-of-assign-operation)

```

When the pointers are incremented, it points to the next element in the array. When the pointers are decremented, it points to the previous element in the array. Our system is limited to pointers only, which means that it does not support pointers to pointers and higher orders. When pointers are incremented or decremented, it must point to an array or it will be considered to be a memory violation.

3.5. Enumeration representation

Enumerations are substituted by their integer values in the model. The enumeration in Listing 2 represents all of the options for a gear shift in an autonomous driving car.

This enumeration is represented in our model as a dictionary, where the key is the enumeration string and the value is its integral value. The enumeration values are represented as follows: UNDEFINED_OR_NO_REQUEST = 0; RANGE_FORWARD = 1; RANGE_REVERSE = 2; NO_TORQUE = 3; IMMOBILIZE = 4; MANUAL_GEAR_1 = 1; MANUAL_GEAR_2 = 1; and MANUAL_GEAR_REV = 2.

Listing 2. Enumeration example

```

1 enum Range_selection_request {
2     UNDEFINED_OR_NO_REQUEST,
3     RANGE_FORWARD,
4     RANGE_REVERSE,
5     NO_TORQUE,
6     MANUAL_GEAR_1 = RANGE_FORWARD,
7     MANUAL_GEAR_2 = RANGE_FORWARD,
8     MANUAL_GEAR_REV = RANGE_REVERSE
9 };

```

3.6. H-CRSM model example

The code in Listing 3 represents a simple function that controls the motion of a self-driving car in the automotive industry. This function represents the disabled autonomous mode state.

Listing 3. Function example

```

1 static dir_motion_control_state_t do_state_disabled(const bool is_auto,
2     const direction_motion_direction_t direction_platform_curr)
3 {
4     dir_motion_control_state_t local_out_state;
5     if (!is_auto)
6     {
7         local_out_state = DIR_MOTION_STATE_DISABLED;
8     }
9     else if (!is_dir_motion_direction_valid(&direction_platform_curr))
10    {
11        counter_reset(&fault_wait_counter);
12        local_out_state = DIR_MOTION_STATE_FAULT;
13    }
14    else
15    {
16        state_machine_const_direction = direction_platform_curr;
17        local_out_state = DIR_MOTION_STATE_CONST_DIRECTION;
18    }
19
20    return local_out_state;
21 }
22 inline bool is_dir_motion_direction_valid(const
23     direction_motion_direction_t* ptr){
24     return *ptr != DIR_UNKNOWN;

```

First, it checks whether there is a request to disable the autonomous mode (which is represented by function parameter variable *is_auto*); if the condition is true, then the autonomous vehicle state is disabled. Second, if there is no autonomous-disable request and the requested direction of motion is not valid, then the autonomous vehicle state is fault. Third, if there is no autonomous-disable request and the requested direction of motion is valid, then the autonomous vehicle state is a constant direction to define the new gear shift state of the car. The H-CRSM diagram is shown in Figure 6.

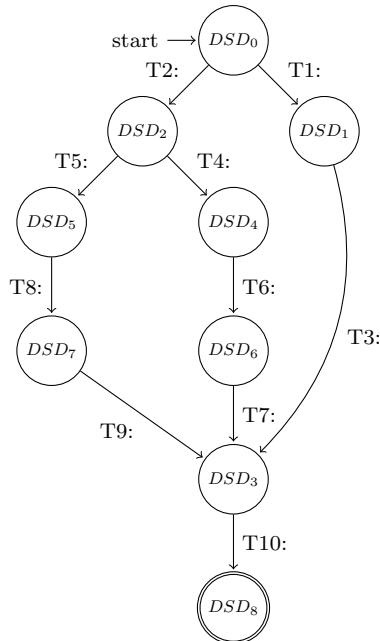


Figure 6. HCRSM diagram for code in Listing 3

The states are represented by circles, and the transitions are represented by directed arrows from one state to another. The commands are shown on each transition. To save space, the command on each transition is described in Table 1. Part of the Extensible Markup Language (XML) is shown in Listing 4. It focuses on the definition of the `do_state_disabled` function in our model, its return values, the variables initialized in this function, and their read and write accesses.

The XML variable initialization part shows that the function has two parameters: `is_auto` and `direction_platform_curr`. The function also has one local variable: `local_out_state`. There is a static variable, which is defined in the *static_global* machine whose name is `state_machine_const_direction`. The read and write accesses for each variable as well as the access transition Identification Number (ID) are shown in the XML.

Table 1
Commands on transitions for code in Figure 6

Transition num.	Command
T1	is_auto == false?
T2	is_auto == true?
T3	local_out_auto_state = DIR_MOTION_STATE_DISABLED
T4	*(&direction_platform_curr) == DIR_UNKOWN?
T5	*(&direction_platform_curr) != DIR_UNKOWN?
T6	counter_reset(&fault_wait_counter)
T7	local_out_auto_state = DIR_MOTION_STATE_FAULT
T8	state_machine_const_direction = dir_platform_curr
T9	local_out_auto_state = DIR_MOTION_STATE_CONST_DIRECTION
T10	return_stmt(local_out_state)

Listing 4. H-CRSM for the code in Listing 3 in XML format

```

1  <machine name = "global_extern" returnType = "void">
2  <container type = "function">
3  <variableInitializations>
4  <varInitNode name = "state_machine_const_direction" type = "
   direction_motion_direction_t">
5  <is_func_param>true</is_func_param>
6  <accessInstance accessType = "WRITE" transition_id = "T8"/>
7  </varInitNode>
8  ...
9  </variableInitializations>
10 ...
11 </machine>
12 <cfile name = "motion_management.c">
13 ...
14 <machine name = "do_state_disabled" returnType = "
   direction_motion_cotrol_state_t">
15 <container type = "function">
16 <variableInitializations>
17 <varInitNode name = "is_auto" type = "const bool">
18 <is_func_param>true</is_func_param>
19 <accessInstance accessType = "READ" transition_id = "T1"/>
20 <accessInstance accessType = "READ" transition_id = "T2"/>
21 </varInitNode>
22 <varInitNode name = "direction_platform_curr" type = "const
   direction_motion_direction_t">
23 <is_func_param>true</is_func_param>
24 <accessInstance accessType = "READ" transition_id = "T4"/>
25 <accessInstance accessType = "READ" transition_id = "T5"/>
26 <accessInstance accessType = "READ" transition_id = "T8"/>
27 </varInitNode>
28 <varInitNode name = "local_out_state" type = "
   direction_motion_cotrol_state_t">
29 <is_func_param>false</is_func_param>
30 <accessInstance accessType = "WRITE" transition_id = "T3"/>
31 <accessInstance accessType = "WRITE" transition_id = "T7"/>
32 <accessInstance accessType = "WRITE" transition_id = "T9"/>
33 </varInitNode>
34 </variableInitializations>
35 ...

```


4. Proposed analyzer

An introduction to the analysis system with the explanation on a simple case study can be found in our paper [4]. The focus of the proposed analysis system is the analysis of the UE equation. Equation `inp1>threshold_val && out1<0` is an example of a UE equation, as `inp1` is a system input and `out1` is a system output. The meaning of the UE equation is that the value of the `out1` signal cannot go beyond zero, given that input signal `inp1` is greater than a threshold value. Building the analysis tree is the process of breaking down all of the variables in the UE equation until a constant value or system input is reached (when neither can be broken into further elements). The ultimate goal of the proposed analyzer is to traverse the analysis tree to find scenarios in the code that validate the UE equation to prove that the code acts in a faulty manner in these corresponding paths. The system requirements of the input system are used to define the UE equation.

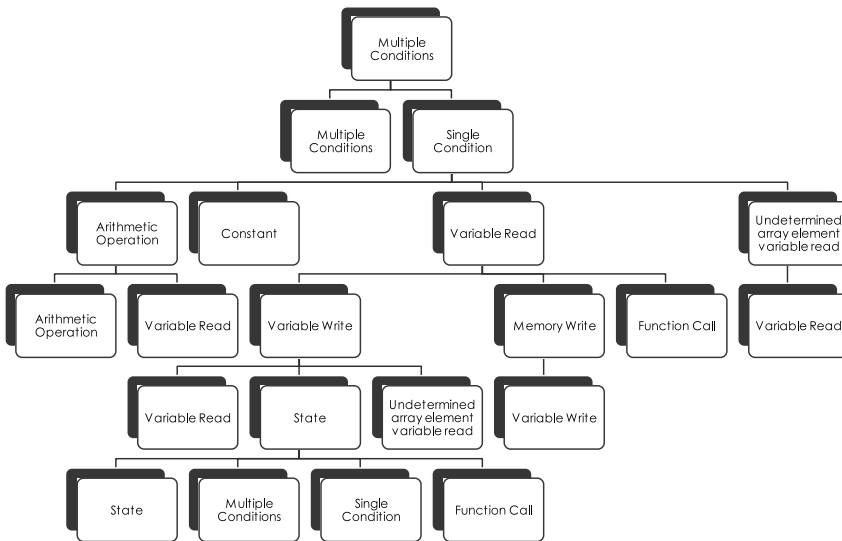


Figure 7. UE element diagram

Figure 2 shows the stages of the proposed analysis system. There are ten different types of UE elements as shown in Figure 7, which are discussed from Section 4.1 through Section 4.10. Section 4.11 discusses the building process of the analysis tree, and Section 4.12 discusses the generation of the hazard scenarios by traversing the generated analysis tree.

4.1. Multiple-condition UE element

A multiple-condition UE element must contain one of the following operators:

- C-and operator (`&&`),
- C-or operator (`||`).

It has two types of children:

- multiple-condition UE element,
- single-condition UE element (Section 4.2)

In UE equation $\text{var1}>4 \ \&\& \ \text{var2}<\text{var1} \ || \ \text{var3}<5$, operator $||$ is the multiple-condition element that has two children. Element $\text{var1}>4 \ \&\& \ \text{var2}<\text{var1}$ is the first child, and element $\text{var3}<5$ is the second child.

Algorithm 12 represents how a multiple-condition element is analyzed to generate its children elements.

Algorithm 12 Multiple-condition element algorithm

```

1: procedure ANALYZEMULTIPLECONDITIONSELEMENT
2:   this-elm.val  $\leftarrow$  get-lowest-priority-elm
3:   if this-element-is-visited-before then
4:     exit-procedure
5:   mark-this-element-as-visited
6:   children-ue-elements-list  $\leftarrow$  analyze-to-generate-children-elements
7:   for each child-ue-element in children-ue-elements-list do
8:     child-ue-element.analyze-ue-equation

```

Function **get-lowest-priority-elm** is responsible for finding the lowest-priority element to be the root value of this element. Therefore, if we have an and-operator and or-operator in the same input string value as in the example above, then the value of the root element will be the or-operator (because it has a lower priority than the and-operator). Function **analyze-to-generate-children-elements** is responsible for generating two children UE-elements. The first UE-child is the string on the LHS of the root element value, and the second UE-child is the string on the RHS of the root element value. In Lines 7 and 8, the algorithm loops through each child and analyzes it to generate the rest of the analysis tree.

4.2. Single-condition UE element

A single-condition UE element must contain one of the following operators:

- C-greater than operator ($>$),
- C-greater than or equal to operator ($>=$),
- C-less than operator ($<$),
- C-less than or equal to operator ($<=$),
- C-equality operator ($==$),
- C-not equal to operator ($!=$).

This can be analyzed into one of the following elements:

- arithmetic expression UE element (Section 4.3),
- variable read UE element (Section 4.4),
- undetermined array variable read UE element (Section 4.5),
- constant UE element (Section 4.9).

In the `var1+var2==arr[i]` UE equation, operator `==` is the single-condition UE element that consists of two children. Element `var1+var2` is the first child, and element `arr[i]` is the second child.

4.3. Arithmetic expression UE element

An arithmetic expression UE element must contain one of the following operators:

- C-assignment operator (`=`),
- C-addition operator (`+`),
- C-subtraction operator (`-`),
- C-multiplication operator (`*`),
- C-division operator (`/`),
- C-arithmetic And operator (`&`),
- C-arithmetic Or operator (`|`),
- C-arithmetic shift-left operator (`<<`),
- C-arithmetic shift-right operator (`>>`).

An arithmetic expression UE element can be analyzed into one of the following UE elements:

- arithmetic expression UE element,
- variable read element (Section 4.4),
- undetermined array variable read UE element (Section 4.5),
- constant UE element (Section 4.9).

In `var1*var2+var3`, the `+` operator represents the arithmetic expression UE element that consists of two elements. Element `var1*var2` represents the first child, and element `var3` represents the second child. Because of the precedence of arithmetic operators in C-language, the multiplication operator has a higher priority than the addition operator. This is why the value of the arithmetic expression UE element in the previous example is the addition operator and not the multiplication operator.

4.4. Variable read UE element

A variable read UE element is the element that abides to the pattern of the variable definition in C-language.

Its children can be one of the following UE elements:

- variable write UE element (Section 4.6),
- memory write UE element (Section 4.7),
- function call UE element return (Section 4.10).

It may not be analyzed into any children if there is no valid path between the variable read-access instance and all of the write accesses to this variable as discussed in the H-CRSM model. A child is added to this UE element for each valid path from the read access to one of the write accesses to the same variable.

Algorithm 13 describes how a variable read element is analyzed. In Line 2, it gets all the write-access instances for the variable. From Line 3 to the end of the algorithm, it loops through each write-access element for this variable. In Line 4, it gets all possible paths from the read-access instance to the write-access instances. A path is defined as a list of transitions that has no writes to the variable in any transitions in that path. This path should begin from the write-access instance transitions and end at this read-access instance transition. In Lines 5–6, it checks that at least one path is found from the read-access instance to this write-access instance; if the condition is true, then it will add this write-access instance as a child with a specific sequence of paths.

Algorithm 13 Variable read element algorithm

```

1: procedure ANALYZEVARIABLEREADELEMENT
2:   write-access-instances  $\leftarrow$  get-var-write-access-instances
3:   for each a-write-access-instance in write-access-instances do
4:     paths  $\leftarrow$  get-paths(a-write-access-instance)
5:     if paths.size > 0 then
6:       add-child(a-write-access-instance, paths)

```

Figure 6 shows the H-CRSM model for the code in Listing 3. When analyzing variable read UE element `local_out_state` on transition T10, it has more than one write access. This is why the value of the variable read UE element is the multiple condition Or-operator. Transition T3 shows the first write-access instance, which is `DIR_MOTION_STATE_DISABLED` when path $\{T1, T3, T10\}$ is traversed. Transition T7 shows the second write-access instance, which is `DIR_MOTION_STATE_FAULT` when path $\{T2, T4, T6, T7, T10\}$ is traversed. Transition T9 shows the third write-access instance, which is `DIR_MOTION_STATE_CONST_DIRECTION` when path $\{T2, T5, T8, T9, T10\}$ is traversed.

If a transition has command `*statePtr = GEAR_NO_TRANSITION` (where `statePtr` is a pointer that points to variable `currState`), then it has only one child when analyzing the read-access instance `currState`, which is memory write element `GEAR_NO_TRANSITION`.

If a transition has command `sensorReading = getLidarSensorReading()`, then it will generate a function call write element `getLidarSensorReading()` when analyzing read-access instance `sensorReading`, which means that the values that can be written to the `sensorReading` variable come from the return values of function `getLidarSensorReading()`.

4.5. Undetermined array UE element variable read

This element defines an array element whose index is not known until run time. `lidarBuf[idx]` is an example of an array variable read element whose index is not known until run time, as `idx` can take any value within the array size range.

This element is analyzed into `n` multiple condition elements (Section 4.1), as `n` is the number of possible array indices for this array, which is any number within the

following range: (0 ... ARRAY_SIZE - 1); i.e., this element is analyzed into **n** elements separated by **or-statements**: (lidarBuf[0] && idx == 0) || (lidarBuf[1] && idx == 1) || (lidarBuf[ARRAY_SIZE - 1] && idx == ARRAY_SIZE - 1).

4.6. Variable write UE element

This element is always a child of a variable read UE element (Section 4.4). The analysis of this UE element consists of two children. A state UE element is the first child that represents the source state of the write-access transition. The following UE elements are candidates for the second child:

- arithmetic expression UE element (Section 4.3),
- variable read UE element (Section 4.4),
- undetermined array UE element variable read (Section 4.5),
- constant UE element (Section 4.9).

When analyzing variable write element `dir_platform_curr` that is the production element when analyzing variable read element `state_machine_const_direction` (which is accessed on transition T8 in Figure 6), it consists of two children. The state UE element DSD_5 that represents the source state of access transition T8 is the first child. Variable read UE element `dir_platform_curr` represents the second child.

When analyzing variable write element `DIR_MOTION_STATE_DISABLED` (which is accessed on transition T3 in Figure 6), it consists of two children. The state UE element DSD_1 that represents the source state of access transition T3 is the first child. Constant UE element `DIR_MOTION_STATE_DISABLED` represents the second child.

If the variable write element is `arr[i]`, it consists of two children elements. The state UE element that represents the source state for the write-access transition is the first child. The undetermined array variable read element `arr[i]` is the second child.

4.7. Memory write UE element

Memory write element is the write to a memory location through a pointer. `*p = var;` is an example of a memory write element, where `var` is the value written using the pointer. It is always a child of a variable read UE element (Section 4.4).

This element is analyzed into variable write element (Section 4.6).

There must be a path from a transition where a pointer is pointing to the address of a variable and another transition that has a memory write to this location without any other writes to that pointer in between so that the memory write is considered to be a write element to the pointee variable.

Figure 8 shows pointer `ptr` that points to `var`. The write to the memory location can be substituted by a write-access element to `var` if and only if the program does not pass through State D. This element is analyzed into a write element to `var` with a list of states that the path must pass through (like State B in this case) to make sure that `ptr` points to `var` and a list of states that the path cannot pass through (like State D in this case).

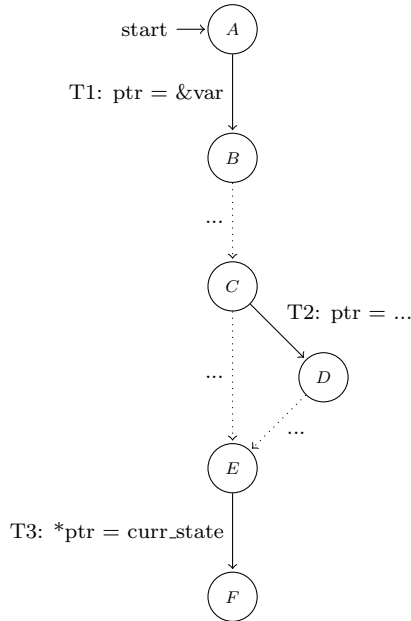


Figure 8. Memory write example

4.8. State UE element

This element can be a child of another state UE element. It can also be the child of a variable write UE element. It has no children if there are no input transitions to this state. It has many children (one for every input transition) if there are no guard conditions on those transitions. The state UE element is analyzed into exactly two children elements in the case of a guarded transition. The guarded condition represents the first child, and the transition's source state represents the second child. It may be analyzed into the following:

- state UE element,
- multiple condition UE element (Section 4.1),
- single condition UE element (Section 4.2).

If state element DSD_1 is analyzed in Figure 6, state DSD_0 will be its only child.

If state element DSD_3 is analyzed, its analysis consists of three children state UE elements. State UE element DSD_7 represents the first child. State UE element DSD_6 represents the second child. State element UE DSD_1 represents the third child.

If state element DSD_2 is analyzed, it consists of two children. Single condition element `is_auto==true?` is the first child. State element DSD_0 is the second child.

4.9. Constant UE element

This element represents any constant value in C-language. It may be characters, numbers, or even C-language reserved words. It is not analyzed into any other children.

4.10. Function call UE element

We can analyze a function call to do the following:

- get the return values of the function. The analysis of each function is done independently, and the generated output scenarios for each function are saved for further use. Each return statement is the beginning state of the function analyzer;
- get the conditions and time steps for each path,
- get the paths that have writes to a global variable,
- get the paths that do not have writes to a global variable.

The code in Listing 3 shows a function called `do_state_disabled` that has three paths and returns a different value for the variable `local_out_state` for each path. If we are analyzing this function to get the return values for this function, then we will get three scenarios (as follows):

- `DIR_MOTION_STATE_DISABLED && is_auto == false.`

This means that the return value is `DIR_MOTION_STATE_DISABLED` and the condition that must be valid for the function to return this value is `is_auto = false.`

- `DIR_MOTION_STATE_FAULT && is_auto == true && is_dir_motion_direction_valid(direction_platform_curr) == false.`

This means that the return value is `DIR_MOTION_STATE_FAULT` and the conditions that must be valid for the function to return this value are `is_auto = true` and `is_dir_motion_direction_valid(direction_platform_curr) = false.`

- `DIR_MOTION_STATE_CONST_DIRECTION && is_auto == true && is_dir_motion_direction_valid(direction_platform_curr) == true.`

If we are analyzing this function to get the paths that have writes to global variable `state_machine_const_direction`, then we will get only one scenario `is_auto == true && is_dir_motion_direction_valid(direction_platform_curr) == true.`

If we are analyzing this function to get the paths that have no writes to global variable `state_machine_const_direction`, then we will get two scenarios:

- `is_auto == false,`
- `is_auto == true && is_dir_motion_direction_valid(direction_platform_curr) == false.`

4.11. Analysis tree construction

Algorithm 14 [4] shows the abstract steps for building an analysis tree. Due to the recursive nature of the construction of the analysis tree process, elements can be visited multiple times. To save space and time, the UE element that is traversed once is saved to be used later without the need to visit it again.

Algorithm 14 Analysis tree construction algorithm [4]

```

1: procedure ANALYZEUEEQUATION
2:   if this-element-is-visited-before then
3:     exit-procedure
4:   mark-this-element-as-visited
5:   children-ue-elements-list  $\leftarrow$  analyze-ue-element-based-on-its-type
6:   for each child-ue-element in children-ue-elements-list do
7:     child-ue-element.analyze-ue-equation

```

An analysis tree built for the code in Listing 3 to get the possible return values of the function is shown in Figure 9. Numbers are added to differentiate the tree nodes. The hazard equation contains the return value of the function, which is *local_out_state*. This also includes the source state that leads to the return statement of the function. This is why the return statement is $DSD3\mathcal{E}\mathcal{E}local_out_state$.

Node (0) is the multiple condition element ($DSD3\mathcal{E}\mathcal{E}local_out_state$) that has two children. Node (1) is the first child, which is state element (DSD3). Node (2) is the second child, which is variable read element *local_out_state*.

Three children are generated when analyzing Node (1). Node (3) is the first child, which is state element $DSD7$. Node (4) is the second child, which is state element $DSD6$. Node (5) is the third child, which is multiple condition element $DSD1$.

Node (9) is the state element $DSD5$ that is generated when analyzing Node (3). The value of Node (9) is changed to $\mathcal{E}\mathcal{E}$, and it is analyzed to generate two children UE elements. Node (19) is the first child, which is single condition element ($*(\mathcal{E}direction_platform_curr) != DIR_UNKOWN$). Node (20) is the second child, which is state element $DSD2$.

The value of Node (19) is changed to $!=$, and it is analyzed to generate two children UE elements. Variable read element *direction_platform_curr* is the first child. Constant value DIR_UNKOWN is the second child.

Node (25) is a leaf element, as there is no write to this variable read element. Node (26) is a leaf element, as it is a constant element.

The value of Node (20) is changed to $\mathcal{E}\mathcal{E}$, and it is analyzed to generate two children. Node (27) is the first child, which is single condition element $is_auto != 0$; this means that the condition in Line 5 in Listing 3 is not satisfied. Node (28) is the second child, which is state element $DSD0$.

The value of Node (27) is changed to $!=$, and it is analyzed to generate two children. Node (33) is the first child, which is variable read element *is_auto*. Node (34) is the second child, which is constant element 0.

Node (33) is a leaf element, as the variable *is_auto* has no write accesses. Node (34) is a leaf element, as it is a constant element.

Node (28) is a leaf element, as state element $DSD0$ does not have any input transitions.

The state element $DSD4$ that is Node (10) is generated when Node (4) is analyzed.

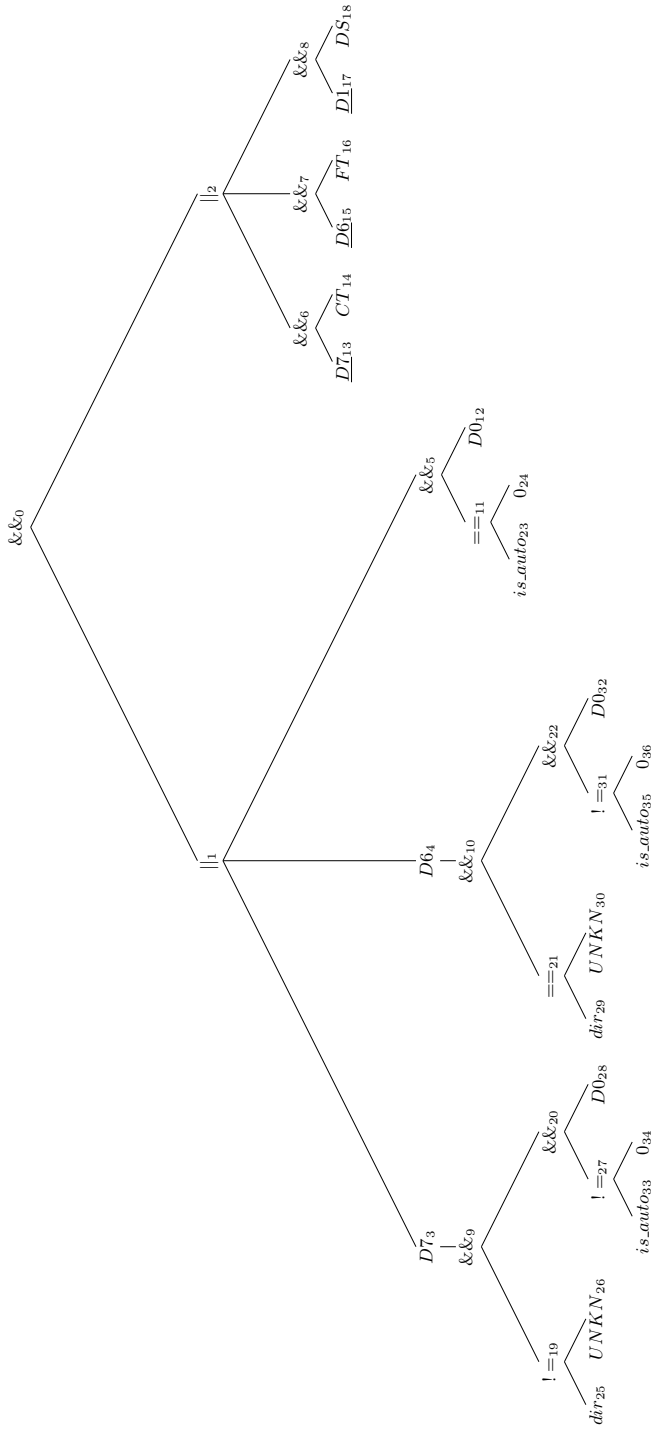


Figure 9. Analysis tree for code in Listing 3 when analyzing function to get return value [4]

The value of Node (10) is changed to $\mathcal{E}\mathcal{E}$ when it is analyzed to generate two children. Node (21) is the first child, which is single condition element *direction_platform_curr == DIR_UNKNOWN* Node (22) is the second child, which is state element *DSD2*.

Node (21) is analyzed into two children, and its value is changed to $\mathcal{E}\mathcal{E}$. The first child (Node 29) is variable read element *direction_platform_curr*. The second child (Node 30) is constant element *DIR_UNKNOWN*. Nodes (29,30) are leaf elements as described before.

The value of Node (22) is changed to $\mathcal{E}\mathcal{E}$ when it is analyzed to generate two children elements. Node (31) is the first child, which is single condition element *is_auto != 0*. Node (32) is the second child, which is state element *DSD0*.

Two children elements are generated when analyzing Node (31). Node (35) is the first child, which is variable read element *is_auto*. Node (36) is the second child, which is constant element *0*. Nodes (35, 36) are leaf elements as described before.

The value of Node (5) is changed to $\mathcal{E}\mathcal{E}$, and it is analyzed to generate two children. Node (11) is the first child, which is single condition element *is_auto == 0*. Node (12) is the second child, which is state element *DSD0*.

The value of Node (11) is changed to $\mathcal{E}\mathcal{E}$, and it is analyzed to generate two children. Node (23) is the first child, which is variable read element *is_auto*. Node (24) is the second child, which is constant element *0*.

The value of Node (2) is changed to (11), and it is analyzed to generate two children elements. Node (6) is the first child, which is variable write element(*DSD7* $\mathcal{E}\mathcal{E}$ *DIR_MOTION_STATE_CONST_DIRECTION*). Node (7) is the second child, which is variable write element(*DSD6* $\mathcal{E}\mathcal{E}$ *DIR_MOTION_STATE_FAULT*). Node (8) is the third child, which is variable write element(*DSD1* $\mathcal{E}\mathcal{E}$ *DIR_MOTION_STATE_DISABLE*).

The value of Node (6) is changed to $\mathcal{E}\mathcal{E}$, and it is analyzed to generate two children. Node (13) is the first child, which is state element *DSD7*. Node (14) is the second child, which is constant element *DIR_MOTION_STATE_CONST_DIRECTION*. The value of Node (7) is changed to $\mathcal{E}\mathcal{E}$, and it is analyzed to generate two children elements. Node (15) is the first child, which is state element *DSD6*. Node (16) is the second child, which is constant element *DIR_MOTION_STATE_FAULT*.

The value of Node (8) is changed to $\mathcal{E}\mathcal{E}$, and it is analyzed to generate two children elements. Node (17) is the first child, which is state element *DSD1*. Node (18) is the second child, which is constant element *DIR_MOTION_STATE_DISABLE*.

The nodes computed before are underlined in the figure.

4.12. Analysis tree traversal

Three paths are shown in the H-CRSM model in Figure 6. Each transition takes one unit of time to execute for simplicity. Consider that the time at state *DSD₀* is T; then, it would be T+1 at state *DSD₂* and T+2 at state *DSD₅*. The model is traversed in reverse order as described in Section 4.11 when the UE equation is analyzed. The

time on the transition where the analysis process begins is T . This means that, if it begins at transition T_{10} , then T is the time at state element DSD_3 . The time at source states DSD_7 , DSD_6 , and DSD_1 is $T-1$.

Algorithm 15 [4] describes the traversal process of the analysis tree.

Algorithm 15 Traversing analysis tree algorithm [4]

```

1: procedure COMPUTESCENARIOS(this-element)
2:   save-status-for-scenario-string
3:   if this-element is-a state-element then
4:     this-element.time  $\leftarrow$  parent-element.time + access-transition.time
5:   else
6:     this-element.time  $\leftarrow$  parent-element.time
7:   if this-element has no children then
8:     if this-element is-a state-element then
9:       add-to-scenario-string(true)
10:    else
11:      add-to-scenario-string(this-element.val)
12:    if this-element is-a lhs-child-for-and-operator then
13:      add-to-scenario-string(and-operator)
14:      compute-scenarios(this-element.parent-and-operator.rhs-child)
15:    else if parent-and-operator is-a lhs-child-for-and-operator then
16:      add-to-scenario-string(and-operator)
17:      next-node  $\leftarrow$  parent-and-operator.parent-and-operator.rhs-child
18:      compute-scenarios(next-node)
19:    else
20:      add-scenario-to-list-of-scenarios
21:    else if this-element.val = or-operator then
22:      for each child in this-element.children do
23:        restore-status-for-scenario-string
24:        compute-scenarios(child)
25:    else if this-element.val = and-operator then
26:      this-element.lhs-child.parent-and-operator  $\leftarrow$  this-element
27:    else
28:      this-element.lhs-child.parent-and-operator  $\leftarrow$  parent-and-operator
29:      compute-scenarios(this-element.lhs-child)
30:    restore-status-for-scenario-string

```

Three possible scenarios are generated when the analysis tree in Figure 9 is traversed:

- `is_auto@(T-2) == 0 && DIR_MOTION_STATE_DISABLED`, when transitions $\{T_1, T_3, T_{10}\}$ in Figure 9 are traversed in the following order: $\{23, 11, 24, 5, 12, 0, 17, 8, 18\}$.
- `direction_platform_curr@(T-3) != DIR_UNKNOWN && is_auto@(T-4) != 0 && DIR_MOTION_STATE_CONST_DIRECTION` when passing through transitions $\{T_2, T_5, T_8, T_9, T_{10}\}$ in Figure 9. The meaning of the `direction_platform_curr@(T-3)` value in the generated scenario is the actual value of variable `direction_platform_curr` at time (T-3). The meaning of the `is_auto@(T-4)` value in the generated scenario is the actual value of variable `is_auto` at time (T-4).

The traversal of the analysis tree in Figure 9 in an order of $\{25, 19, 26, 33, 27, 34, 20, 28, 13, 6, 14\}$ generates the previous scenario. State element DSD_0 is substituted by `true` when the state element has no children.

When traversing the analysis tree, the following hazard scenario is generated: `direction_platform_curr != DIR_UNKNOWN && is_auto != 0 && DSD0 && DIR_MOTION_STATE_CONST_DIRECTION`.

- `direction_platform_curr@(T-4) == DIR_UNKNOWN && is_auto@(T-5) != 0 && DIR_MOTION_STATE_FAULT` when passing through transitions {T2, T4, T6, T7, T10} in Figure 9 in the following order: {29, 21, 30, 10, 35, 31, 36, 22, 32, 0, 15, 7, 16}.

We will concentrate on the first generated hazard scenario in Figure 9, which is (`is_auto@(T-2) == 0` $\mathcal{E}\mathcal{E}$ `DIR_MOTION_STATE_DISABLED`). The traversing of the tree begins from the root, which is (Node 0). The algorithm checks the children for (Node 0) from left to right; i.e., the next traversed child is (Node 1). (Node 1) is an or-operator, which means that each child of this node will produce a separate path. For simplicity, consider that (Nodes 3,4) are already traversed. The algorithm traverses (Node 5), which is an and-operator. The algorithm traverses (Node 11), then (Node 23). (Node 23) is a leaf element, so its value is added to *scenario-string*. *scenario-string* now has the value of *is.auto*. The algorithm adds the value of its parent, which is (Node 11). The value of *scenario-string* is (`is_auto ==`). The algorithm finds that the parent of the current-node (Node 11) is an and-operator, so it adds (`&&`) to *scenario-string*, which is the value of (Node 11), and traverses the RHS of the and-operator, which is (Node 12). (Node 12) is a leaf state element, and its value is substituted by *true*. The value of *scenario-string* is (`is_auto == 0` $\mathcal{E}\mathcal{E}$ *true*). The algorithm finds that (Node 11) is the RHS of the and-operator in (Node 5), and it searches for the first parent-and-operator for (Node 5). The first parent-and-operator is (Node 0). The and-operator (Node 5) is in the LHS of the parent-and-operator (Node 0), so the algorithm traverses the RHS of (Node 0), which is (Node 2). (Node 2) is an or-operator, so it traverses its first child (Node 6) then its child (Node 13). The algorithm finds that (Node 13) is a state statement and has not been visited before when traversing the tree, so this path is neglected. The same happens when traversing (Nodes 7, 15). The algorithm traverses (Node 8) then its child (Node 17). It finds that state element (Node 17) is traversed before in (Node 5), so the algorithm continues to traverse this path, and it traverses (Node 18), which is the RHS of and-operator (Node 8). (Node 8) is constant-element `DIR_MOTION_STATE_DISABLED`. The algorithm adds it to *scenario-string* to be (`is_auto == 0` $\mathcal{E}\mathcal{E}$ *true* $\mathcal{E}\mathcal{E}$ `DIR_MOTION_STATE_DISABLED`). (Node 18) is the RHS of and-operator (Node 8), which is the RHS of and-operator (Node 0), which is the root, so the traversing of the path ends and the value inside *scenario-string* is added to the *scenarios-list* that contains all of the hazard scenarios. The final value of this hazard scenario after simplification is (`is_auto == 0` $\mathcal{E}\mathcal{E}$ `DIR_MOTION_STATE_DISABLED`).

Table 2 compares the proposed work and the previous related work. The related work focuses on the system and the software level of the software life cycle to model and verify the system.

The proposed system focuses on modeling the implemented software in C-language and verifying the implementation correctness with the help of the undesirable hazard equation analysis using the built formal model.

Table 2
Comparison between proposed work and others

Paper	Depends on requirements	Depends on software model	Depends on system model	Depends on implementation
Proposed system	✗	✗	✗	✓
[27]	✓	✓	✗	✗
[14]	✗	✗	✓	✗
[33]	✓	✓	✓	✗
[34]	✗	✗	✓	✗

5. Case studies

This section summarizes the model and the analysis systems on two real-world examples. It shows how our system is capable of detecting errors in the input C-project and how it is able to generate real-time test cases to prove that the system may be faulty under specific circumstances (if any exist). The case studies are described in Section 5.1 and Section 5.2.

5.1. Case study 1 – auto pilot

Listing 5 contains part of the code for the *state_run* function that calls the *do_state_disabled* function. There is a purposeful typo in the function in Listing 5 in Line 4 that indicates that the condition should be *STATE_DISABLED*, not *STATE_AUTO_FORWARD*. The UE-equation given by the user is (*curr_state != STATE_AUTO_DISABLED && output_state == DIR_MOTION_STATE_DISABLED*). This means that the hazard that we do not want to happen in the system is that the system outputs *STATE_AUTO_DISABLED* for the *out_state* signal when input signal *curr_state* is *DIR_MOTION_STATE_DISABLED*. Variable *curr_state* is a system input. Variable *output_state* is a system output.

Listing 5. Case Study 1 example

```

1 dir_motion_control_state_t state_run(const bool is_auto, const
   direction_motion_direction_t direction_platform_curr)
2 {
3   ...
4   if(curr_state == STATEAUTO_FORWARD)
   output_state = do_state_disabled(is_auto, direction_platform_curr);
6   else if(curr_state == STATEAUTO_FORWARD)

```

```

7     output_state = do_state_auto_forward(is_auto ,
8         direction_platform_curr);
9     ...
10    return output_state;
11 }

```

The system will analyze this UE-equation as described in Section 4.12. When analyzing the *do_state_disabled* function call in Line 5 in Listing 5 using the above UE-equation, the hazard scenarios are as follows:

- $curr_state \neq STATE_AUTO_DISABLED \ \&\& \ is_auto == 0 \ \&\& \ DIR_MOTION_STATE_DISABLED == DIR_MOTION_STATE_DISABLED$
Element *output_state* in the UE-equation is substituted by *DIR_MOTION_STATE_DISABLED*, which is the LHS of the (*==*) operator.
- $curr_state \neq STATE_AUTO_DISABLED \ \&\& \ direction_platform_curr \neq DIR_UNKOWN \ \&\& \ is_auto \neq 0 \ \&\& \ DIR_MOTION_STATE_CONST_DIRECTION == DIR_MOTION_STATE_DISABLED$
Element *output_state* in the UE-equation is substituted by *DIR_MOTION_STATE_CONST_DIRECTION*. When simplifying this equation, its value is *false* because of the term (*DIR_MOTION_STATE_CONST_DIRECTION == DIR_MOTION_STATE_DISABLED*).
- $curr_state \neq STATE_AUTO_DISABLED \ \&\& \ direction_platform_curr == DIR_UNKOWN \ \&\& \ is_auto \neq 0 \ \&\& \ DIR_MOTION_STATE_FAULT == DIR_MOTION_STATE_DISABLED$
Element *output_state* in the UE-equation is substituted by *DIR_MOTION_STATE_FAULT*. When simplifying this equation, its value is *false* because of the term (*DIR_MOTION_STATE_FAULT == DIR_MOTION_STATE_DISABLED*).

The only scenario that does not evaluate to false is $curr_state \neq STATE_AUTO_DISABLED \ \&\& \ is_auto == 0 \ \&\& \ DIR_MOTION_STATE_DISABLED == DIR_MOTION_STATE_DISABLED$. The table generator module takes the responsibility for generating the possible values for each system input in the hazard equation that may cause the hazard to occur as shown in Figure 2 given the hazard scenarios from the analyzer and the classes for each system input. The classes for a system input are the range of values that can be given to this variable. In this case, the user can define the range of values for *curr_state* as {*STATE_AUTO_DISABLED*, *STATE_AUTO_FORWARD*, *STATE_AUTO_REVERSE*, ...} and the range of values for *is_auto* as {0, 1}. Given this information, the system will substitute *curr_state* by the values that will make the hazard equation happen. In this case, the value of *curr_state* is *STATE_AUTO_DISABLED*, and the value of *is_auto* is 0; then, the hazard equation after substitution is (*STATE_AUTO_DISABLED != STATE_AUTO_DISABLED && is_auto == 0 && DIR_MOTION_STATE_DISABLED == DIR_MOTION_STATE_DISABLED*), which evaluates to true.

5.2. Case study 2 – auto parking

Listing 6 contains part of the code for an automatic parking module for a car. The first function (*ManageA_P_ClientMode*) is called from the cyclic function of the automatic parking module that executes every 25 milliseconds. This is responsible for updating the automatic parking data and state machine. It is also responsible for reinitializing the module if the input electric system suffers from any failure.

Listing 6. Case Study 2 example

```

1 static void ManageA_P_ClientMode(void)
2 {
3     if(A_P.inputs.electric_system == ELEC_SYS_STOP)
4     {
5         A_P_Init();
6     }
7     else
8     {
9         UpdateA_P_Data();
10        UpdateA_P_StateMachine();
11        ...
12    }
13 }
14
15 void A_P_Init(void)
16 {
17     ResetStateMachine();
18     InitA_P_OutputData();
19     ...
20 }
21
22 static void ResetStateMachine(void)
23 {
24     ...
25     A_P.ctrl_state = A_P_FINISH;
26     ...
27 }
28
29 static void UpdateA_P_Data(void)
30 {
31     bool is_obstacle_detected = IsObstacleDetected();
32     ...
33     A_P.is_finished =
34     ( ( (A_P.ctrl_state == A_P_FINISH)
35       ||(A_P.ctrl_state == A_P_ABORT)
36       ||(A_P.ctrl_state == USER_GIVEN_CTRL) )
37       &&(A_P.inputs.brake_pressure >= 80u)
38       &&(A_P.is_vehicle_stopped != 0 ) )? 1:0;
39     ...
40 }

```

The second function (*A_P_Init*) is the initialization function for the automatic parking module that is responsible for initializing all of the global data as well as

resetting the timers and output buffers. The third function (*ResetStateMachine*) is responsible for resetting the state machine controls in the case of initializing the automatic parking module. The fourth function (*UpdateA_P_Data*) is responsible for updating the output data after performing the logic necessary for the automatic parking module.

Let us consider that the hazard that we do not want to happen in our code is that the automatic parking module is finished while the brakes are not pressed, the vehicle is not stopped, or the automatic parking control state is not one of the following: *finished*, *aborted*, or *user given control*.

The variable responsible for determining the completion state of the automatic parking module is defined in Line 33. There is a purposeful typo in Line 38 where we type $A_P.is_vehicle_stopped == 0$; however, for the automatic parking to be completed, the condition should be negated, and the $==$ should be replaced by $!=$. The UE equation is $A_P.is_finished == 1 \ \&\& \ A_P.is_vehicle_stopped == 0$, which means that the automatic parking is completed and the vehicle is not in a stationary state. Figure 10 shows the H-CRSM diagram for the *ManageA_P_ClientMode* function.

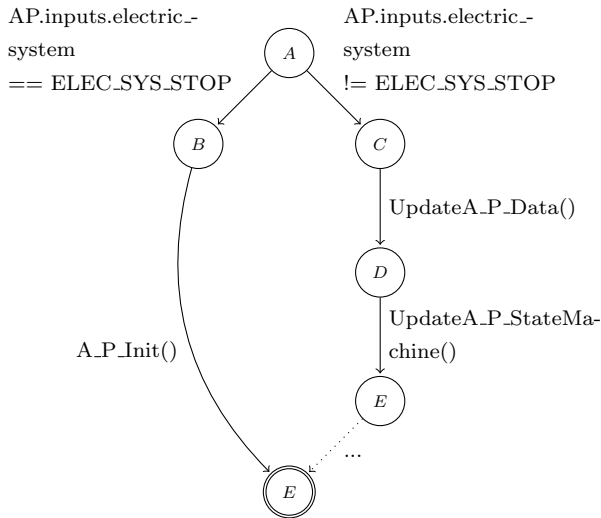


Figure 10. H-CRSM for *ManageA_P_ClientMode* function in Listing 6

Figure 11 shows the H-CRSM diagram for the *A_P_Init* function. Figure 12 shows the H-CRSM diagram for the *ResetStateMachine* function. Figure 13 shows the H-CRSM diagram for the *UpdateA_P_Data* function.

Listing 7 shows part of the XML output for the H-CRSM model for the code in Listing 6. This focuses on the *globalMachine* that contains the definitions for all of the global variables in the *automatic_parking.c* file. The initialization node for *A_P.is_finished* variable is defined in Lines 6 through 9. It does not have any read accesses,

and it has only one write access on transition $T3$ inside the *UpdateA_P_Data* function. The initialization node for the *A_P.ctrl_state* variable is defined in Lines 10 through 14. This has a read access on transition $T3$ inside the *UpdateA_P_Data* function. It also has only one write access on transition $T2$ inside the *ResetStateMachine* function. The initialization node for the *A_P.inputs.brake_pressure* variable is defined in Lines 19 through 22. This has only one read access on transition $T3$ inside the *UpdateA_P_Data* function. The initialization node for the *A_P.is_vehicle_stopped* variable is defined in Lines 23 through 26. This has only one read access on transition $T3$ inside the *UpdateA_P_Data* function.

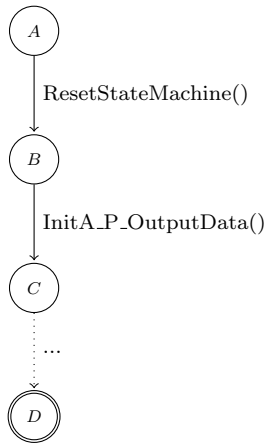


Figure 11. H-CRSM for *A_P_Init* function in Listing 6

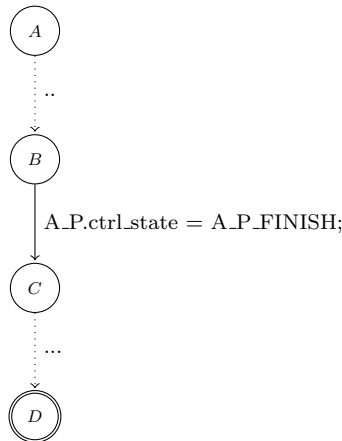


Figure 12. H-CRSM for *ResetStateMachine* function in Listing 6

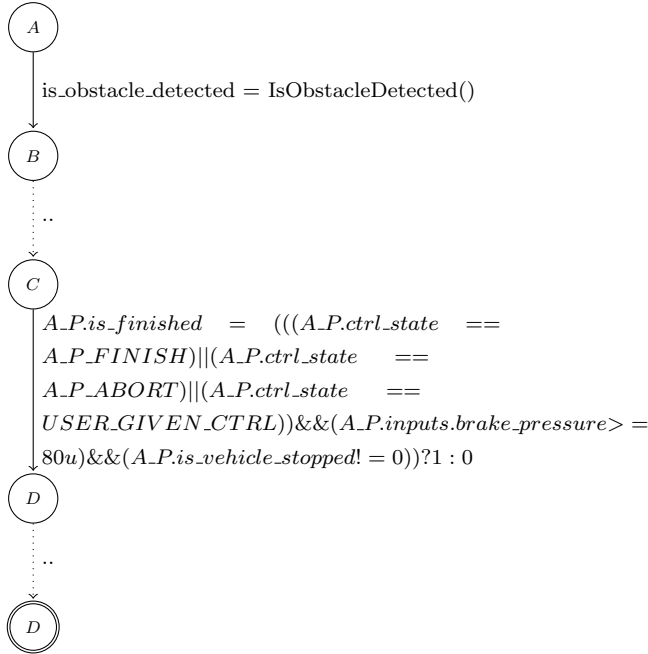


Figure 13. H-CRSM for UpdateA_P_Data function in Listing 6

Listing 7. H-CRSM for the code in Listing 6 in XML format

```

1 <cfile name = "automatic_parking.c">
2   ...
3   <machine name = "globalMachine" returnType = "void">
4     <container type = "function">
5       <variableInitializations>
6         <varInitNode name = "A_P.is_finished" type = "bool">
7           <is_func_param>false</is_func_param>
8           <accessInstance accessType = "WRITE" transition_id =
9             "UpdateA_P_Data-T3"/>
10          </varInitNode>
11         <varInitNode name = "A_P.ctrl_state" type = "enum
12           AP_ControlState_e">
13           <is_func_param>false</is_func_param>
14           <accessInstance accessType = "READ" transition_id =
15             "UpdateA_P_Data-T3"/>
16           <accessInstance accessType = "WRITE" transition_id =
17             "ResetStateMachine-T2"/>
18          </varInitNode>
19         <varInitNode name = "A_P.inputs.electric_system" type = "
20           enum AP_ElectricSys_e">
21           <is_func_param>false</is_func_param>
22           <accessInstance accessType = "READ" transition_id = "
23             ManageA_P_ClientMode-T1"/>
24          </varInitNode>
25         <varInitNode name = "A_P.inputs.brake_pressure" type = "
26           enum AP_ElectricSys_e">

```

```

20         <is_func_param>false</is_func_param>
21         <accessInstance accessType = "READ" transition_id = "
UpdateA_P_Data-T3"/>
22     </varInitNode>
23     <varInitNode name = "A_P.is_vehicle_stopped" type = "enum
AP_ElectricSys_e">
24         <is_func_param>false</is_func_param>
25         <accessInstance accessType = "READ" transition_id = "
UpdateA_P_Data-T3"/>
26     </varInitNode>
27 </variableInitializations>
28     ...

```

The analysis tree for the code in Listing 6 when analyzing hazard equation $A_P.is_finished == 1 \ \&\& \ A_P.is_vehicle_stopped == 0$ can be found in Figure 15, and the rest of the diagram can be found in Figure 14. The hazard equation is broken into a multiple condition element (Node 0). It is also broken into two children. The first child (Node 1) is logical operation $A_P.is_finished == 1$. The second child (Node 2) is logical operation $A_P.events.is_vehicle_stopped == 0$. (Node 1) is broken into a variable read element $A_P.is_finished$ indicated as (Node 3) and a constant expression indicated as (Node 4). (Node 3) is broken into the variable write element for variable $A_P.is_finished$ expressed as (Node 7). Based on the XML in Listing 7, it has only one write access in function *UpdateA_P_Data*. Its value is substituted by the RHS of the equal operation on transition *T3* in function *UpdateA_P_Data*. (Node 7) is broken into two children. The first child (Node 8) expresses the LHS of the *and-operation*, which is $(A_P.ctrl_state == A_P_FINISH) \ || \ (A_P.ctrl_state == A_P_ABORT) \ || \ (A_P.ctrl_state == USER_GIVEN_CTRL)$. The second child (Node 9) expresses the RHS of *and-operation*, which is $A_P.is_vehicle_stopped! = 0 \ \&\& \ A_P.inputs.brake_pressure \geq 80u$. (Node 8) is broken into two children. The first child (Node 10) expresses the LHS of *or-operation*, which is $A_P.ctrl_state == A_P_FINISH$. The second child (Node 15) expresses the RHS of *or-operation*, which is $(A_P.ctrl_state == A_P_ABORT) \ || \ (A_P.ctrl_state == USER_GIVEN_CTRL)$. (Node 10) is broken into two children. The first child (Node 14) expresses the LHS of the equal operation. The second child (Node 15) expresses constant element A_P_FINISH . (Node 11) is broken into two children. The first child (Node 16) is the logical operation on the LHS of *or-operation*, which is $A_P.ctrl_state == A_P_ABORT$. The second child (Node 17) is the logical operation on the RHS of *or-operation*, which is $A_P.ctrl_state == USER_GIVEN_CTRL$. (Node 16) is broken into two children. The first child (Node 22) is the LHS of the equal operation, which is variable read element $A_P.ctrl_state$. The second child (Node 23) is the RHS of the equal operation, which is constant element A_P_ABORT . (Node 17) is broken into two children. The first child (Node 24) is the LHS of the equal operation, which is variable read element $A_P.ctrl_state$. The second child (Node 25) is the RHS of the equal operation, which is constant element $USER_GIVEN_CTRL$. (Nodes 14, 22, and 24) are variable write element $A_P.ctrl_state$. Their expansion is described in Figure 14. (Node 9) is broken into two children. The first child (Node 12) is logical expression

$A_P.inputs.brake_pressure \geq 80$. The second child (Node 13) is logical expression $A_P.is_vehicle_stopped == 0$. (Node 12) is broken into two children. The first child (Node 18) is variable read element $A_P.inputs.brake_pressure$. The second child (Node 19) is constant element 80 . (Node 18) is not expanded to any other write elements, as the XML for the model shows that it does not have any write accesses through the program, so it is considered to be a system input. (Node 13) is broken into two children. The first child (Node 20) is variable read element $A_P.is_vehicle_stopped$. The second child (Node 21) is constant element 0 . (Node 20) is not expanded to any other write elements for the same reason as discussed for (Node 18). (Node 2) is expanded into two children. The first child (Node 5) is variable read element $A_P.is_vehicle_stopped$. The second child (Node 6) is constant element 0 . (Node 5) is the same element as (Node 20).

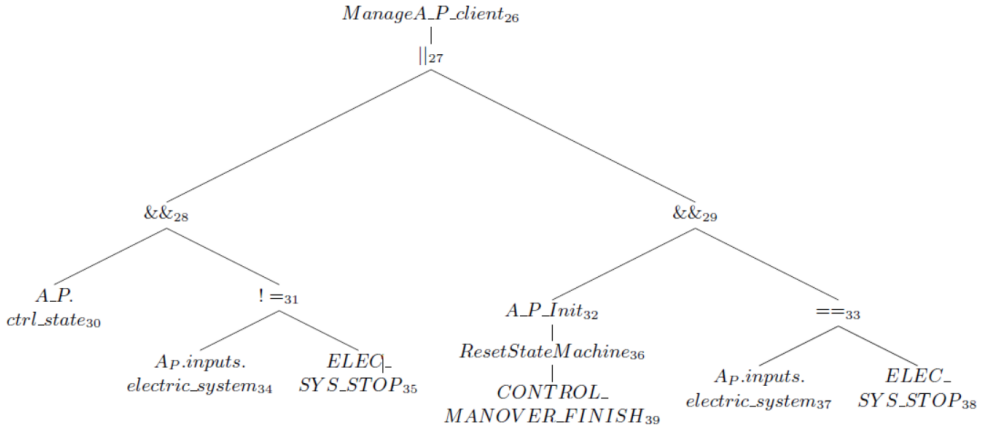


Figure 14. Analysis tree for code in Listing 6 when analyzing *ManageA_P_ClientMode* to get values for global variable *A_P.ctrl_state*

Figure 14 shows the remaining part of the analysis tree in Figure 15, which are the children of (Nodes 14, 22, and 24). It shows the expansion of write access element *A_P.ctrl_state*. As shown in Listing 7, variable *A_P.ctrl_state* has a write access inside the *ResetStateMachine* function on transition 2. The system tries to find a path from current function *UpdateA_P_Data* to function *ResetStateMachine*. The function call graph is shown in Figure 16. *ManageA_P_ClientMode* (MCM) calls the following functions: *A_P_Init* (Init), *UpdateA_P_Data* (UD), and *UpdateA_P_StateMachine* (USM). The *A_P_Init* function calls the following functions: *ResetStateMachine* (RSM) and *InitA_P_OutputData* (InitOD). The *UpdateA_P_Data* function calls function *IsObstacleDetected* (IOD). The function call sequence to get from the read-access to the write-access for variable *A_P.ctrl_state* is {UD, MCM, Init, InitOD}, as shown in Figure 16.

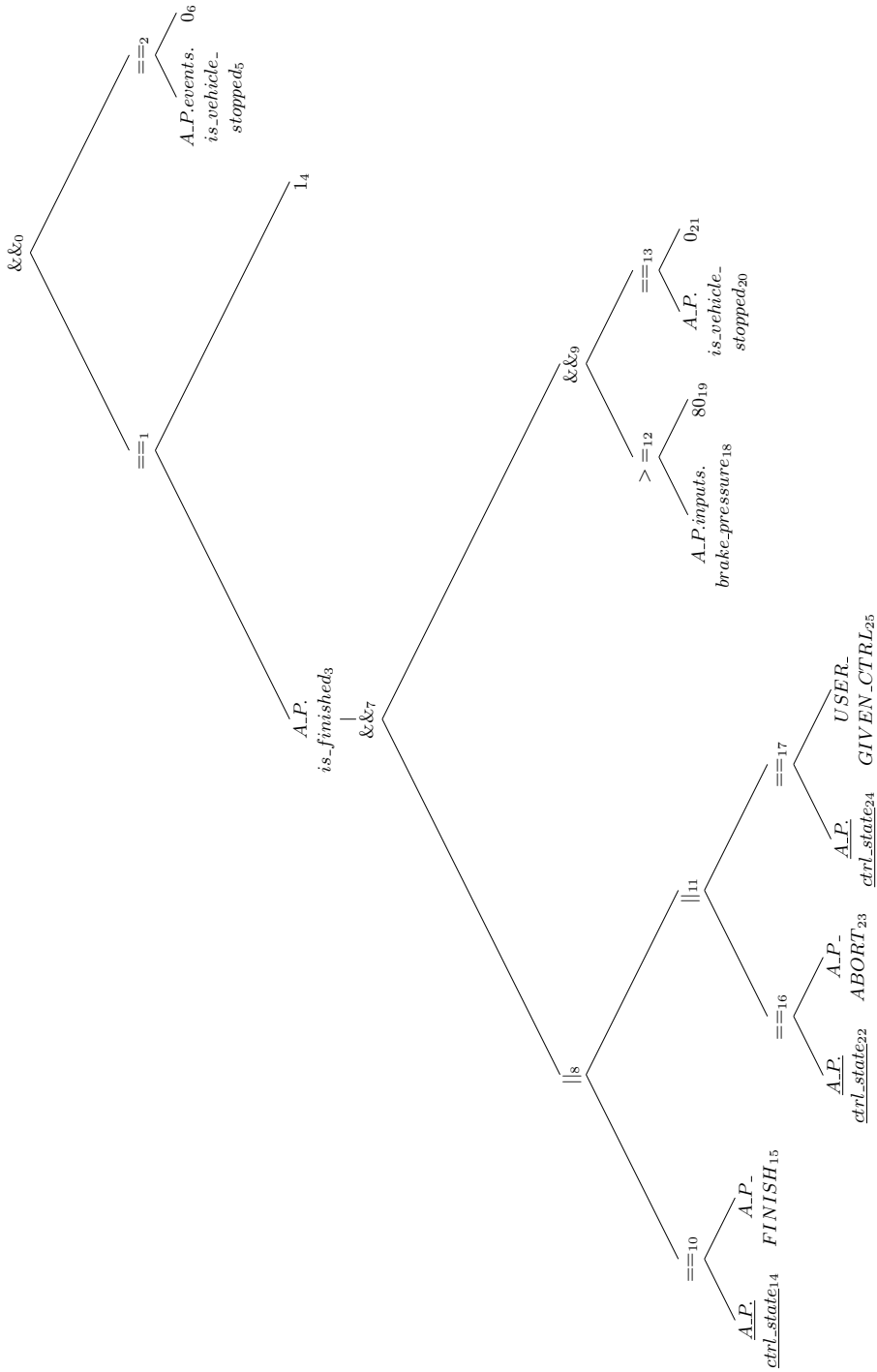


Figure 15. Analysis tree for code in Listing 6 when analyzing UE equation $A.P.is_finished==1 \ \&\& \ A.P.is_vehicle_stopped==0$

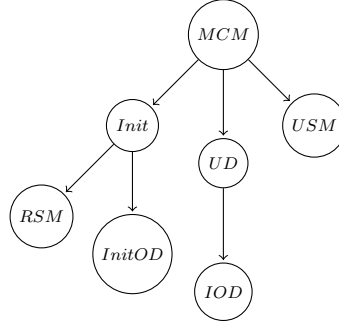


Figure 16. Function call graph for code in Listing 6

(Node 26) is function call element *ManageA_P_client*, which is traversed to find the write accesses to global variable *A_P.ctrl.state*. This is expanded into the logical *or-expression* (Node 27). (Node 27) represents two scenarios. The first is (Node 28), which is the current value of variable *A_P.ctrl.state* when condition *A_P.inputs.electric_system == ELEC_SYS_STOP* is not satisfied. The second is (Node 29), which is the value *A_P.FINISH* that happens when condition *A_P.inputs.electric_system == ELEC_SYS_STOP* is satisfied so that function *ManageA_P_ClientMode* calls function *A_P_Init* that calls function *ResetStateMachine* to update global variable *A_P.ctrl.state* to value *A_P.FINISH*.

When traversing the analysis tree in Figure 14, the following two scenarios are generated:

1. $A_P.ctrl.state \ \&\& \ A_P.inputs.electric_system \ != \ ELEC_SYS_STOP$.
2. $A_P.FINISH \ \&\& \ A_P.inputs.electric_system \ == \ ELEC_SYS_STOP$.

When traversing Figure 14 to get the values for global variable *A_P.ctrl.state* inside function *ManageA_P_clientMode*, the following scenarios are generated:

1. $A_P.ctrl.state \ == \ A_P.FINISH \ \&\& \ A_P.inputs.electric_system \ != \ ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \ >= \ 80 \ \&\& \ A_P.is_vehicle_stopped \ == \ 0$.
2. $A_P.FINISH \ == \ A_P.FINISH \ \&\& \ A_P.inputs.electric_system \ == \ ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \ >= \ 80 \ \&\& \ A_P.is_vehicle_stopped \ == \ 0$.
3. $A_P.ctrl.state \ == \ A_P.ABORT \ \&\& \ A_P.inputs.electric_system \ != \ ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \ >= \ 80 \ \&\& \ A_P.is_vehicle_stopped \ == \ 0$.
4. $A_P.FINISH \ == \ A_P.ABORT \ \&\& \ A_P.inputs.electric_system \ == \ ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \ >= \ 80 \ \&\& \ A_P.is_vehicle_stopped \ == \ 0$.
5. $A_P.ctrl.state \ == \ USER_GIVEN_CTRL \ \&\& \ A_P.inputs.electric_system \ != \ ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \ >= \ 80 \ \&\& \ A_P.is_vehicle_stopped \ == \ 0$.
6. $A_P.FINISH \ == \ USER_GIVEN_CTRL \ \&\& \ A_P.inputs.electric_system \ == \ ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \ >= \ 80 \ \&\& \ A_P.is_vehicle_stopped \ == \ 0$.

After running the simplifier, it gets rid of the fourth scenario, as condition $A_P_FINISH == A_P_ABORT$ will evaluate to false. It gets also rid of the sixth scenario, as condition $A_P_FINISH == USER_GIVEN_CTRL$ will evaluate to false. The final list of the possible hazard equations is as follows:

1. $A_P.ctrl_state == A_P_FINISH \ \&\& \ A_P.inputs.electric_system \neq ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \geq 80 \ \&\& \ A_P.is_vehicle_stopped == 0.$
2. $A_P_FINISH == A_P_FINISH \ \&\& \ A_P.inputs.electric_system == ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \geq 80 \ \&\& \ A_P.is_vehicle_stopped == 0.$
3. $A_P.ctrl_state == A_P_ABORT \ \&\& \ A_P.inputs.electric_system \neq ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \geq 80 \ \&\& \ A_P.is_vehicle_stopped == 0.$
4. $A_P.ctrl_state == USER_GIVEN_CTRL \ \&\& \ A_P.inputs.electric_system \neq ELEC_SYS_STOP \ \&\& \ A_P.inputs.brake_pressure \geq 80 \ \&\& \ A_P.is_vehicle_stopped == 0.$

The case study runs on a Windows machine that has an I7 processor, 16 GB of RAM, and six MB three-level cache. It took three minutes for the modeler and pre-processor to generate the H-CRSM model. The C-code has 29,028 lines of code after pre-processing and resolving all includes and definitions. It took 14 minutes to build the analysis tree, traverse it, and generate the list of scenarios. The system consumes 60 MB of memory during both the modeler and analysis phases.

6. Conclusion

The accidents that still happen nowadays due to buggy software are the trigger of our research on finding hazards that may occur in the implemented systems. The purpose of the work is to model and analyze safety critical systems that work in real-time to generate a list of potential hazard scenarios. The H-CRSM formal model is used to develop a novel modeling approach. It is used to parse real-time systems that are written in ANSI-C, generate the AST, traverse the AST, and generate an H-CRSM model. It extracts important semantics from the input C-code. C-LANG is the best choice to parse the input C-code and generate the AST. The modeling of the functions in the input C-project can be done in parallel due to the hierarchical property of the model. The modeled machines are linked together after the end of the modeling phase. This method is very helpful when modeling large systems. A new approach for hazard analysis is proposed that handles time-critical safety systems. The analyzer module takes as input the H-CRSM model as well as the hazardous equation, which is analyzed by the H-CRSM model. This analysis produces a list of hazardous scenarios that might occur in the ANSI-C input code. Case studies were discussed to support the proposed methodology; namely, the generation of an analysis tree and how the tree is traversed in order to produce a hazards list. The analyzer's strength lies in running it statically without the need to run the code in real-time. It also accurately finds the exact values of the inputs to the system at certain times that would result in hazardous situations. Two case studies are discussed to show how

the implemented system can detect bugs that may occur in the input C-project. The proposed system is not limited to C-Language but could be extended to any other programming language.


References

- [1] 4th International Workshop on CPAchecker, 2019. <https://cpa.sosy-lab.org/2019/> [online; accessed 11.11.2019].
- [2] Andrews J.D., Dunnett S.J.: Event-tree analysis using binary decision diagrams. In: *IEEE Transactions on Reliability*, vol. 49(2), pp. 230–238, 2000. <https://doi.org/10.1109/24.877343>.
- [3] Astrée Static-analysis Tool, 2019. <http://www.astree.ens.fr/> [online; accessed 11.11.2019].
- [4] Bakr A.M., Fouda M.M., Salama M., Alsammak A.K., Yahia H.: Hazard Analysis of Real-time Safety Critical Systems Using Hierarchical Communicating Real-time State Machines Formal Model. In: *2017 12th International Conference on Computer Engineering and Systems (ICCES)*, pp. 628–634, 2017. <https://doi.org/10.1109/ICCES.2017.8275381>.
- [5] Bakr A.M., Fouda M.M., Salama M., Alsammak A.K., Yahia H.: Modeling real-time safety critical systems using hierarchical communicating real-time state machines and c-lang parser. In: *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, pp. 244–251, 2017. <https://doi.org/10.1109/INTELCIS.2017.8260054>.
- [6] Bertolino A.: Software Testing Research: Achievements, Challenges, Dreams. In: *Future of Software Engineering, 2007, FOSE '07*, pp. 85–103, 2007.
- [7] BLAST Static-analysis Tool, 2019. <http://mtc.epfl.ch/software-tools/blast/index-epfl.php> [online; accessed 11.11.2019].
- [8] Coverity, 2019. <https://scan.coverity.com/> [online; accessed 11.11.2019].
- [9] Ericson C.A.: Fault tree analysis. In: *System Safety Conference, Orlando, Florida*, vol. 1, pp. 1–9, 1999.
- [10] Ericson C.A.: *Hazard analysis techniques for system safety*. John Wiley & Sons, 2015.
- [11] Ethiopian Airlines Flight 302 – Wikipedia, The Free Encyclopedia, 2019. https://en.wikipedia.org/wiki/Ethiopian_Airlines_Flight_302 [online; accessed 9.04.2019].
- [12] Fortino G., Furfaro A., Nigro L., Pupo F.: Hierarchical Communicating Real-Time State Machines. In: *Laboratorio di Ingegneria del Software, Dipartimento di Elettronica Informatica e Sistemistica, Universita della Calabria*, pp. 1–12, 2000.
- [13] Furfaro A., Nigro L.: Model Checking Hierarchical Communicating Real-time State Machines. In: *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 1, pp. 354–370, 2005.

- [14] Gario A., Andrews A., Hagerman S.: Testing of safety-critical systems: An aerospace launch application. In: *2014 IEEE Aerospace Conference*, pp. 1–17, 2014. <https://doi.org/10.1109/AERO.2014.6836410>.
- [15] Helix-QAC, 2019. <https://www.perforce.com/products/helix-qac> [online; accessed 11.11.2019].
- [16] Infer, 2019. <https://fbinfer.com/> [online; accessed 11.11.2019].
- [17] Ishimatsu T., Leveson N.G., Thomas J., Katahira M., Miyamoto Y., Nakao H.: Modeling and Hazard Analysis Using STPA. In: *Proceedings of the 4th IAASS Conference, Making Safety Matter*, pp. 19–21, 2010.
- [18] Jenab K., Dhillon B.S.: Stochastic Fault Tree Analysis With Self-loop Basic Events. In: *IEEE Transactions on Reliability*, vol. 54(1), pp. 173–180, 2005. <https://doi.org/10.1109/TR.2004.842087>.
- [19] Johannessen P., Grante C., Alminger A., Eklund U., Torin J.: Hazard analysis in object oriented design of dependable systems. In: *2001 International Conference on Dependable Systems and Networks*, pp. 507–512, 2001. <https://doi.org/10.1109/DSN.2001.941436>.
- [20] Lee, Seshia: *Introduction to Embedded Systems*. MIT Press, 2017.
- [21] Lee E.A.: Cyber Physical Systems: Design Challenges. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369, 2008.
- [22] Leupolz J., Habermaier A., Reif W.: Quantitative and qualitative safety analysis of a hemodialysis machine with S#. In: *Journal of Software: Evolution and Process*, vol. 30(5), p. e1942, 2018. <https://doi.org/10.1002/smr.1942>. E1942 JSME-17-0029.R2.
- [23] Lion Air Flight 610 – Wikipedia, The Free Encyclopedia, 2019. https://en.wikipedia.org/wiki/Lion_Air_Flight_610 [online; accessed 9.04.2019].
- [24] Lutz R.R.: Analyzing software requirements errors in safety-critical, embedded systems. In: *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pp. 126–133, 1993. <https://doi.org/10.1109/ISRE.1993.324825>.
- [25] Macher G., Sporer H., Berlach R., Armengaud E., Kreiner C.: SAHARA: A security-aware hazard and risk analysis method. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 621–624, 2015. <https://doi.org/10.7873/DATE.2015.0622>.
- [26] Polyspace, 2019. <https://www.mathworks.com/products/polyspace.html> [online; accessed 11.11.2019].
- [27] Rao C., Guo J., Li N., Lei Y., Zhang Y.L., Li Y.: Safety-Critical System Modeling in Model-Based Testing with Hazard and Operability Analysis. In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 397–404, 2018. <https://doi.org/10.1109/QRS.2018.00053>.

- [28] Reddy Y.B.: Cloud-Based Cyber Physical Systems: Design Challenges and Security Needs. In: *2014 10th International Conference on Mobile Ad-hoc and Sensor Networks*, pp. 315–322. 2014. <https://doi.org/10.1109/MSN.2014.50>.
- [29] Shaw A.C.: Communicating real-time state machines. In: *IEEE Transactions on Software Engineering*, vol. 18(9), pp. 805–816, 1992.
- [30] Takahashi M., Kosaka R., Nanba R., Anang Y., Watanabe Y.: A study of methodology for securing control software based FMEA-FTA coordination. In: *2016 IEEE/SICE International Symposium on System Integration (SII)*, pp. 144–149, 2016. <https://doi.org/10.1109/SII.2016.7843989>.
- [31] Tesla driver crash with a truck, 2019. <https://electrek.co/2019/03/01/tesla-driver-crash-truck-trailer-autopilot/> [online; accessed 9.04.2019].
- [32] Toyota car recalls – Wikipedia, The Free Encyclopedia, 2010. https://en.wikipedia.org/wiki/2009%E2%80%9311_Toyota_vehicle_recalls [online; accessed 9.04.2019].
- [33] Yoo J., Jee E., Cha S.: Formal Modeling and Verification of Safety-Critical Software. In: *IEEE Software*, vol. 26(3), pp. 42–49, 2009. <https://doi.org/10.1109/MS.2009.67>.
- [34] Zhu D., Yao S.: A Hazard Analysis Method for Software-Controlled Systems Based on System-Theoretic Accident Modeling and Process. In: *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 90–95, 2018. <https://doi.org/10.1109/ICSESS.2018.8663927>.

Affiliations

Ahmed M. Bakr 

Benha University, Faculty of Engineering at Shoubra, Egypt, ahmed.bakr@feng.bu.edu.eg,
ORCID ID: <https://orcid.org/0000-0003-1387-6178>

May Salama

Benha University, Faculty of Engineering at Shoubra, Egypt, may.mohamed@feng.bu.edu.eg

Abdelwahab K. Alsammak

Benha University, Faculty of Engineering at Shoubra, Egypt, asammak@feng.bu.edu.eg

Received: 17.12.2019

Revised: 03.05.2020

Accepted: 03.05.2020