Transformation- and Pattern-based State Machine Mining from Embedded C Code

Andreas Grosche[1], Burkhard Igel[2] and Olaf Spinczyk[3]
[1]*Behr-Hella Thermocontrol GmbH, Hansastraße 40, 59557 Lippstadt, Germany*
[2]*Fachhochschule Dortmund, Sonnenstraße 96, 44139 Dortmund, Germany*
[3]*Universität Osnabrück, Wachsbleiche 27, 49090 Osnabrück, Germany*

Keywords: State Machine Extraction, Model Mining, Reverse Engineering, Program Comprehension, Refactoring.

Abstract: Automated extraction of state machine models from source code can improve comprehension of software system behavior required for many maintenance tasks and reuse in general. Furthermore, it can be used for subsequent automated processing such as refactoring and model-based verification. This paper presents an approach based on normalizing transformations of an input program and a pattern to find state machine implementations in the program and to extract relevant information. The results are used to create state machine models containing states, transitions, events, guards and actions. Fine-grained traceability between the model and the source code enables navigation and refactoring of model elements. We evaluate the approach by applying a prototypical implementation to industrial automotive embedded code and show that 74 % of the expected state machine implementations can be completely identified and 8 % partially.

1 INTRODUCTION

While model-based development, e.g., using Matlab/Simulink, has been established for developing the application software layer of automotive embedded systems, low-level programming languages such as C are preferably used for time-critical and basic software to gain transparency and control over registers, memory, runtime and synchronization (Grossman et al., 2005). There are several reasons for the motivation of reverse engineering such code:

Maintenance. A developer has to comprehend relevant parts of a software system before changes can be applied (von Mayrhauser and Vans, 1995). If the documentation is obsolete or not available, the source code has to be analyzed. Said et al. (Said et al., 2019) show that state machine models extracted from C code help to understand embedded software. In addition, state-based model-driven development has been shown to improve efficiency during maintenance (Ricca et al., 2018).

Evolution. Small state machine implementations that are not explicitly documented due to their simplicity can gradually evolve over time into larger state machines that are difficult to understand.

Reuse. Parts of a software system may be reused in new projects in the original or an adapted form. A good understanding of the interfaces and behavior is required for correct integration.

Verification. Comprehension of the design and implementation must be available to manually derive or automatically generate test cases. Furthermore, finite state verification techniques can help to detect deadlocks, livelocks and race conditions among other properties. These techniques require a formal model.

Validation. A design helps ensuring that the developed system meets the expectations of the customer.

Refactoring. Restructuring a state machine in sequential code, e.g., merging two states, can be a tedious and error-prone task compared to modifying state machine diagrams. In addition, unstructured or disadvantageous implementations may be replaced by more sophisticated realizations, usages of common libraries or frameworks (Fowler, 2013).

Migration. Changing the development methodology to model-based design like

Grosche, A., Igel, B. and Spinczyk, O.
Transformation- and Pattern-based State Machine Mining from Embedded C Code.
DOI: 10.5220/0009769801040115
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 104-115
ISBN: 978-989-758-443-5

mbeddr[1] (Voelter et al., 2019) can make existing code unusable. Since legacy code typically contains valuable expert knowledge evolved over years, a migration of such code may be intended.

State machines are a commonly accepted documentation and implementation technique for the dynamic behavior of reactive systems. They are widely used in embedded systems to react to inputs, for device drivers that handle asynchronous sequences and to subdivide continuous processes into smaller stateful steps processable in discrete time slices. However, low-level C code is too detailed for quick understanding and state machine implementations can be scattered over the program making manual reverse engineering of state machines tedious and error-prone.

The extraction of state machine models from source code is known as *state machine mining* (Said et al., 2018). Existing approaches are either not fully automated, do not provide enough information or provide results that cannot be used for further automated processing like refactoring.

In this paper, we propose an approach to automatically extract a specific subset of UML state machine[2] implementations from source code. The extracted models contain the states, transitions, events, guards and actions of the implementations with traceability between each model element and the source code. These models can be used for program comprehension during maintenance, reuse, verification and validation as well as automated refactoring and migration. To allow the adaptation to different implementation techniques, our approach is based on editable search patterns. The transformation-based approach allows to find implementations with several variations.

Our evaluation using industrial automotive embedded C code shows the soundness of our approach. While the pattern matching is complete, the completeness of the state machine mining depends on the specified pattern and the degree of normalization.

This paper is structured as follows: Existing approaches for state machine mining, normalization and code search are presented in Section 2. Section 3 explains the widely used state machine implementation technique we are interested in. Our approach to find such implementations is introduced in Section 4. The evaluation in Section 5 using industrial automotive embedded code shows that 74 of 100 state machine models could be completely extracted from the source code. Section 6 concludes and gives an outlook.

[1]The mbeddr approach closes the gap between high-level graphical models and low-level C code.

[2]UML sate machines are based on statecharts introduced by Harel (Harel, 1987).

2 RELATED WORK

Dynamic State Machine Mining. Dynamic state machine mining approaches typically instrument the source code of a program to create execution traces or log data that are analyzed to recover a state machine from the program (Xiao et al., 2013; Ammons et al., 2002). Many techniques are based on *state merging* (Biermann and Feldman, 1972). The required information can also be fetched from unit test executions (Xie et al., 2006). These approaches tend to generate partial state machines without information about how variables are modified. Walkinshaw and Hall (Walkinshaw and Hall, 2016) address this issue by extending the dynamic analysis with a technique based on genetic programming to extract additional information. However, dynamic approaches are not applicable to all embedded systems since code instrumentation alters the execution time and could therefore break the typical hard real-time constraints (Thums and Quante, 2012). Our approach circumvents instrumentation by using static analysis techniques.

Static State Machine Mining. Static approaches are commonly based on text, program representations such as the abstract syntax tree, control flow analysis, data flow analysis, symbolic execution or concolic testing.

Prywes et al. (Prywes and Rehmet, 1996) propose a human guided approach that creates a state for each condition as well as transitions that represent a sequence of operations. The result is close to a control flow graph and does not reflect stateful behavior maintained over multiple function calls.

A technique that incorporates Cpp2XMI to find nested `if` and `switch` statements is proposed by van den Brand et al. (van den Brand et al., 2008). Although the implementation supports simple syntactic variations, it has to be explicitly extended to support further ones. Our approach supports many more variations implicitly using preceding normalization.

Somé and Lethbridge (Somé and Lethbridge, 2002) find state variables by naming conventions and use a *state variable definition graph* to identify and reverse engineer state machine implementation idioms. Our approach is independent of names.

Walkinshaw et al. (Walkinshaw et al., 2008) discover state transitions and actions with respective source code using symbolic execution. However, the states have to be identified manually for this approach.

Jiresal et al. (Jiresal et al., 2011) use heuristic-based abstractions to extract statecharts. Their approach is specific to state machine implementation techniques that use variables for the communication

with the context. The proposed abstractions may hinder automated refactoring.

Bandera (Corbett et al., 2000) automatically extracts state machines from Java code intended to be used for model checking. The results are not meant to be human comprehensible or to be used, e.g., for refactoring.

The proposal of Wang et al. (Shaohui Wang et al., 2012) uses symbolic execution to extract state machines from graphical user interface software where screens are represented as states. This approach is restricted to user interface software.

Kung et al. (Kung et al., 1994) use symbolic execution to extract object state behaviors from object-oriented C++ source code. Sen and Mall (Sen and Mall, 2016) apply this approach to Java and extend it to address some shortcomings. Said et al. (Said et al., 2018) further extend these approaches by using concolic testing to overcome limitations in symbolic execution regarding complex constraint solving and allow user-defined constraints. They further adapt the approach for industrial automotive embedded systems developed with the programming language C and let the user interactively specify constraints to reduce the complexity of the mined state machine models.

Approaches using symbolic execution or concolic testing typically produce incomplete models, e.g., with missing events and actions, that are not suitable for further automated processing like refactoring. In contrast, our work addresses a subset of the state machines that can be found by the stated approaches that can be automatically identified and refactored. In addition, the patterns of our approach can be adapted to find, e.g., framework-based implementations that cannot be found by symbolic execution or concolic testing.

Our approach is based on the work of Knor et al. (Knor et al., 1998) where a pattern is matched with the source code of a program to identify possible code fragments that implement a state machine and may be manually refactored. In contrast to the proposed enhanced string pattern search that requires user interaction, our approach is based on graph representations known from program analysis to extract all required information from the source code for automated model construction.

Normalization. Necula et al. (Necula et al., 2002) propose transformations to normalize C code to the *C intermediate language (CIL)*. This subset of C reduces the complexity of analysis tools since less language concepts have to be considered.

Xu and Chee (Xu and Chee, 2003) propose transformation-based diagnosis for comparing student programs with reference solutions for automatic grading in the context of programming tutoring systems. The student and model programs, i.e., reference solutions, are transformed into a subset of the programming language using *semantics-preserving transformations*. The transformed programs are represented as annotated augmented object-oriented program dependence graphs that are used for program matching. However, the model program does not allow to extract information from the matched student program via metavariables.

We use the concepts of CIL (Necula et al., 2002) and the transformation-based diagnosis (Xu and Chee, 2003) to normalize the state machine search pattern and the input program to cover multiple programs with semantics-preserving variations using only one pattern. We extend the approaches by introducing a fine-grained traceability.

Code Search. Several tools and techniques have been proposed for code search with placeholders for the extraction of information from a program for further processing. We prefer pattern-based code search with patterns based on the input program language over queries like ASTLOG (Crew, 1997) or A* (Ladd and Ramming, 1995) to provide easier access to the pattern language. Existing techniques range from textual to lexical, syntactic and graph-based matching.

SCRUPLE (Paul and Prakash, 1994) analyzes the attributed syntax tree of an input program and offers wildcards as well as concepts in the pattern language to cover some syntactic variations. However, these have to be explicitly selected.

Coccinelle (Brunel et al., 2009) is a transformation tool for Linux collateral evolutions. It uses temporal logic for matching patterns on control-flow paths and supports metavariables. While Coccinelle matches simple variations such as swappable operands using isomorphisms, it does not match more complex variations such as `if` and `switch` statements. Moreover, Coccinelle does not support matching of nested conditional statements with arbitrary nesting depth.

We adopt the ideas of SCRUPLE and Coccinelle and extend them by normalizing the inputs to match more syntactic variations, e.g., `if` statements match equivalent `switch` and `case` statements. In addition, our approach supports matching of recursive fragments such as nested `if` statements with arbitrary nesting depth. Our approach furthermore provides match results as a data structure with access to data analysis results and different program representations for the subsequent model extraction.

3 STATE MACHINE IMPLEMENTATION TECHNIQUES

Several techniques are used to implement state machines in source code. These include nested `switch` statements, state tables, object-oriented state design patterns as well as combinations and variations (Samek, 2009). This paper focuses on nested `switch` statements because this technique is often used due to its simplicity, although it is difficult to read as it may be scattered over the code and may contain large nesting depths. Other implementation techniques can be supported by adapting the pattern and extraction (see Section 4).

Listing 1 shows a simplified example of a state machine implementation of an embedded system's device driver that uses a serial peripheral interface (SPI) to send a request to an external integrated circuit (IC) and waits for an acknowledgment from the IC in the form of an external interrupt. The write operation and waiting for the acknowledgment are asynchronous processes that are handled by the state machine. In addition, a timeout handling is performed that aborts the operation after 100 ms of missing write completed or acknowledgment event.

A state variable holds the current state of the state machine (see Line 8). The initialization value `IDLE` represents the initial state. The functions `Request`, `Cyclic10ms`, `WriteCompleted` and `AckReceived` represent the events the state machine can process. Calling the `Request` function (see Line 12) causes the reset of a `time` counter (see Line 15), the start of the asynchronous SPI write operation (see Line 16) and a transition to the *WRITING* state (see Line 17). The underlying SPI driver calls the `WriteCompleted` function (see Line 36) on completion of the SPI transmission which causes the reset of the `time` counter and a state change to the *WAITING_FOR_ACK* state. An interrupt handler calls the `AckReceived` function (see Line 43) if the external IC raises the acknowledgment signal. As a consequence, the state machine transits to the *IDLE* state.

To implement the timeout handling, the `Cyclic10ms` function (see Line 22) is called by the operating system with a period time of 10 ms. If the state machine is in the `WRITING` or `WAITING_FOR_ACK` state, the timeout handling is performed. The `time` counter is incremented by 10 according to the function call period time. If the `time` counter is greater than or equal to 100 ms, a timeout is detected and notified calling the `NotifyTimeout` function. In addition, a transition to the *TIMEOUT* state is performed.

The example shows that `if` statements can be used in combination with `switch` statements, the state machine implementation can be scattered over several functions and guards are optional.

Listing 1: Running example of a state machine implementation in C code. Critical sections required to handle exclusive access to the state variable from main and interrupt context as well as other details are omitted for simplicity.

```
1  typedef enum {
2   IDLE,
3   WRITING,
4   WAITING_FOR_ACK,
5   TIMEOUT
6  } StateType;
7
8  StateType state = IDLE;
9  int time;
10 int isTimeout;
11
12 void Request() {
13  switch (state) {
14  case IDLE:
15   time = 0;
16   SPIWrite(5);
17   state = WRITING;
18   break;
19  }
20 }
21
22 void Cyclic10ms() {
23  switch (state) {
24   case WRITING:
25   case WAITING_FOR_ACK:
26    time += 10;
27    isTimeout = time >= 100;
28    if (isTimeout) {
29     NotifyTimeout();
30     state = TIMEOUT;
31    }
32    break;
33  }
34 }
35
36 void WriteCompleted() {
37  if (WRITING != state)
38   return;
39  time = 0;
40  state = WAITING_FOR_ACK;
41 }
42
43 void AckReceived() {
44  if (state == WAITING_FOR_ACK)
45   state = IDLE;
46 }
```

4 STATE MACHINE MINING

We propose a transformation- and pattern-based approach as shown in Figure 1 to support several varia-

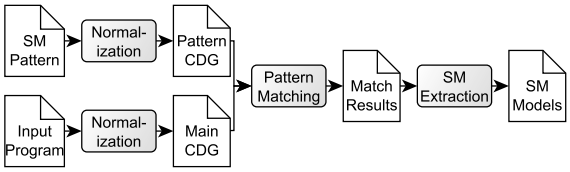

Figure 1: The state machine mining pipeline consisting of the phases *normalization*, *pattern matching* and *state machine extraction*.

tions in the state machine implementation and adaptation of patterns to cover different state machine implementation techniques. A *state machine pattern* (SM pattern) and the source code of an *input program* are normalized in the *normalization* phase. The *pattern matching* phase uses the resulting *control dependence graphs* (pattern and main CDG) to provide *match results* containing code fragments that implement state machines. The *state machine extraction* (SM extraction) phase uses the results to create *state machine models* (SM models) containing states, transitions, events, guards and actions. This section describes the pattern definition and the phases in detail using the nested `switch` state machine implementation technique from Listing 1 as running example.

4.1 Pattern Definition

A pattern that matches the nested `switch` implementation technique from the running example must provide the following information as identified by Knor et al. (Knor et al., 1998):

- The *source code fragments* that make up the state machine. These fragments can be scattered over the program, e.g., split into multiple functions as shown in Listing 1. The pattern is designed to cover all relevant fragments.
- The *state space* consisting of all possible states of the state machine. The pattern matches statements where a constant expression is assigned to a state variable as well as expressions where a state variable is compared to a constant expression using the equality operator `==`. The constant expressions are interpreted as concrete states.
- The *state variable* that holds the current state of the state machine. This variable is determined by using the variable in the statements from the aforementioned state space identification.
- The *event space* consisting of all events the state machine accepts. Somé and Lethbridge (Somé and Lethbridge, 2002) identified two implementation techniques for event handling. *Single routine* implementations use a single function that handle events using a `switch` statement comparable to guard handling. *Multiple routine* implementations provide a distinct function for each event. Calling such function lets the state machine handle the event. The pattern in Listing 4 covers a single function. To address the multiple routine implementation from the running example (see Listing 1), the results of multiple matches are merged in the extraction phase described in Section 4.4.
- The state *transitions*, *guards* and *actions* are matched by the pattern using metavariables. The values of the metavariables are used in the extraction phase (see Section 4.4) to create the transitions and add the guards and actions.

To explain the concepts of the pattern and matching we incrementally develop a pattern to match the state machine implementation from Listing 1 in three steps. Starting with a small pattern that matches only one code fragment in the first step, the second step matches all but one and the last step matches all code fragments.

4.1.1 First Step

The first pattern (see Listing 2) is designed to match the simplest code fragment, namely the contents of the `AckReceived` function (see Listing 1 Line 43). This code fragment changes the state to `IDLE` if the current state is `WAITING_FOR_ACK`.

A pattern is defined in the programming language C with additional semantics. The code fragments to be matched are wrapped in functions called *pattern declarations*. Each pattern declaration annotated with `@main` is matched with each function of the input program. A pattern declaration matches a code fragment from the input program if all statements and expressions match. Variable declarations are ignored during matching since the location of the declarations are considered arbitrary by default and matching of the usages is sufficient[3].

The `if` statement of the pattern (see Listing 2 Line 6) is matched with the `if` statement of the `AckReceived` function from the input program (see Listing 1 Line 44). *Metavariables* can be used to match variables and constant expressions by using the built-in types `varmetavar` and `constmetavar`, respectively. The pattern in Listing 2 declares the `stateVar` metavariable to match variables in the input program. Each reference to `stateVar` in the pattern has to match with a variable reference to the same variable in the input program. Regarding the `AckReceived` function, `stateVar` matches the `state` variable. The `stateConst0`

[3]Variable declaration locations to be matched can be constrained by annotating the respective declarations in the pattern.

and `stateConst1` metavariables match the constant expressions `WAITING_FOR_ACK` and `IDLE`, respectively. Distinct metavariables have to be used since the constant expressions evaluate to different values. The values of metavariables can be used for further processing in the extraction phase.

Listing 2: Pattern of the first step that matches the contents of the `AckReceived` function from the input program shown in Listing 1.

```
1 /** @main */
2 void PStateCheckAndChange() {
3  varmetavar stateVar;
4  constmetavar stateConst0;
5  constmetavar stateConst1;
6  if (stateVar == stateConst0)
7   stateVar = stateConst1;
8 }
```

4.1.2 Second Step

The normalization preceding the matching transforms the `switch` statement in function `Request` (see Listing 1 Line 13) to an `if` statement and the *not equal* operator of function `WriteCompleted` (see Line 37) to an *equal* operator. These normalized conditions are already matched by the condition `stateVar == stateConst0` from the first pattern. However, the pattern matches `if` statements where the *then* branch contains only the assignment of a new state to the state variable. Code fragments containing additional statements in the *then* branch like in functions `Request`, `Cyclic10ms` and `WriteCompleted` are not covered. The pattern shown in Listing 3 extends the first pattern to match these statements which are interpreted as actions executed during the state transition.

Sequential statements can be moved to a new pattern declaration. The new pattern declaration can be referenced using the built-in `Match` statement and providing a *match expression*. The contents of the referenced pattern declaration are matched instead of the `Match` statement. Match expressions are inspired by regular expressions where character classes like `[A-Z]` are replaced with names of pattern declarations, e.g., `[PatternName]` to reference a pattern declaration. These references can be embedded in named groups of the form `(?<GroupName>[PatternName])`. The code fragment matched by the referenced pattern declaration is available in the extraction phase using the specified group name.

In the pattern shown in Listing 3, the assignment of a new state to the state variable is moved to the new pattern declaration `PStateChange` (see Line 12). To match an arbitrary number of statements with an

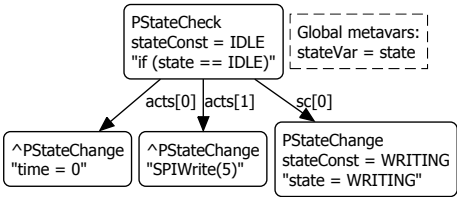

Figure 2: The *pattern instances tree* shows the instances of the pattern declarations from Listing 3 created during matching with the `Request` function from the running example (see Listing 1).

eventual state change, the `PStateChange` pattern declaration is referenced twice. The first reference (see Line 7) uses the negation operator `^` and the quantification `*` known from regular expressions to match an arbitrary number of any statement but a state change. The second reference matches the state change, i.e., an assignment of a constant expression to the state variable.

The pattern matcher creates pattern instances for each successfully matched pattern declaration. This includes the entry pattern declaration annotated with `@main` as well as referenced pattern declarations. The resulting *pattern instances tree* from matching the pattern from Listing 3 with the `Request` function from Listing 1 is shown in Figure 2.

Metavariables can be declared locally or globally. Local metavariables are declared within a pattern declaration (see Line 5 and Line 13). Local metavariables of different pattern instances may contain different values. Global metavariables are declared outside a pattern declaration (see Line 1). A global metavariable referenced by different pattern instances must have the same value for a successful match. For example, the `stateVar` global metavariable referenced in `PStateCheck` and `PStateChange` always match the variable `state` from the input program (see Figure 2).

Listing 3: Pattern of the second step that matches the contents of all functions but Cyclic10ms from the input program shown in Listing 1.

```
1  varmetavar stateVar;
2
3  /** @main */
4  void PStateCheck() {
5   constmetavar stateConst;
6   if (stateVar == stateConst) {
7    Match("(?<acts>[^PStateChange]*)");
8    Match("(?<sc>[PStateChange])");
9   }
10 }
11
12 void PStateChange() {
13  constmetavar stateConst;
14  stateVar = stateConst;
15 }
```

4.1.3 Third Step

The pattern from the second step does not match the `Cyclic10ms` function since it does not support state changes that are *indirectly* control dependent on state checks, i.e., it does not support guards.

Listing 4: Pattern of the third step that matches the contents of all functions from the input program shown in Listing 1.

```
1  varmetavar stateVar;
2
3  /** @main @level 0 @coverage
       partialtoplevel */
4  void PStateCheck() {
5   constmetavar stateConst;
6   if (stateVar == stateConst)
7    Match("(?<guardWActs>[PGuardWActs]) |
         (?<sc>[PStateChangeWActs])");
8   else
9    Match("(?<check>[PStateCheck]?)");
10 }
11
12 void PGuardWActs() {
13  Match("(?<acts>[^PGuard]*)");
14  Match("(?<guard>[PGuard])");
15 }
16
17 void PGuard() {
18  exprmetavar guard;
19  if (guard)
20   Match("(?<scwa>[PStateChangeWActs])");
21  else
22   Match("(?<guard>[PGuard]) | (?<stmts>[
         Stmt]*)");
23 }
24
25 void PStateChangeWActs() {
26  Match("(?<acts>[^PStateChange]*)");
27  Match("(?<sc>[PStateChange])");
28 }
29
30 void PStateChange() {
31  constmetavar stateConst;
32  stateVar = stateConst;
33 }
```

The pattern in Listing 4 circumvents this limitation by introducing the `PGuardWActs` and `PGuard` pattern declarations. In addition, the two match statements used to match actions and a state change are moved to the new pattern declaration `PStateChangeWActs`. Moreover, the `if` statement in `PStateCheck` allows an optional *else* branch by specifying a reference to the `PStateCheck` pattern declaration and using the `?` quantifier. This recursive reference matches state checks nested in the *false* branch of a state check. Since the normalization phase transforms `switch` statements to nested `if` statements, the `switch` statement of the `Cyclic10ms` function is matched by this pattern.

The `PStateCheck` pattern declaration expects either a guard (`PGuardWActs`) or a state change with optional actions (`PStateChangeWActs`) in the *then* branch. The alternation is denoted using the *or* operator `|` known from regular expressions.

A guard as declared in `PGuard` is an `if` statement with an arbitrary condition. The metavariable `guard` of type `exprmetavar` matches any expression. The *then* branch consists of a state change with optional actions (`PStateChangeWActs`) and the *else* branch consists of a nested guard (`PGuard`) or arbitrary statements (`Stmt`). `Stmt` is a built-in pattern declaration that matches any statement.

4.2 Normalization

Searching for state machine implementations according to the nested `switch` statement technique, possible code variations have to be considered. For example, a `switch` statement can be replaced by `if` statements and the operands of the equality operator `==` can be swapped. Variations between two programs that change the computational behavior while maintaining the computational result are known as *semantics-preserving variations* (Xu and Chee, 2003).

The normalization phase uses different program representations to transform the input program and the pattern to remove semantics-preserving variations. These *semantics-preserving transformations* are based on Xu's and Chee's (Xu and Chee, 2003) approach for comparing student programs with a specimen program for programming tutoring systems as well as the *C intermediate language* (Necula et al., 2002). We adapt these approaches by using a subset of the proposed transformations that still allow for tracing back to the original code and the identification of state machine implementations.

4.2.1 Abstract Syntax Tree

The first step of the normalization phase is parsing the source code and representing it as an abstract syntax tree (AST). This step involves textual transformations performed by the C preprocessor, tokenization and parsing to construct the AST.

The AST represents the source code without syntactic details such as formatting. Subsequent transformations remove syntactic variations from the program. These include, e.g., the replacement of `switch` statements with nested `if` statements. Conditions of `if` statements are normalized by swapping the *then* and *else* branches if necessary. For example, the not equal operator `!=` in Listing 1 Line 37 is normalized to an equal operator `==` by introducing an empty *else* branch and swapping the *then* and *else* branches.

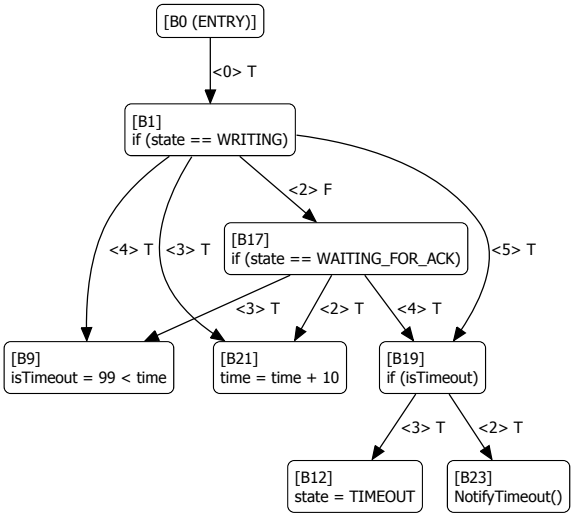

Figure 3: The *control dependence graph* of function `Cyclic10ms` from the running example shows the control dependences between the statements.

4.2.2 Control Flow Graph

The second step of the normalization is the creation of a control flow graph (CFG) based on the AST. The CFG abstracts the sequential source code lines by making the control flow explicit and hiding `goto` and `label` statements.

4.2.3 Control Dependence Graph

The control dependence graph (CDG) is constructed using the control flow graph by computing the post-dominators, constructing the post-dominator tree and determining the control dependences as proposed by Ferrante et al. (Ferrante et al., 1987). We use a simplified variant where the nodes of a CDG represent statements and the edges represent the control conditions on which the statements are executed.

Figure 3 shows the CDG of the `Cyclic10ms` function from the running example (see Listing 1). Nodes containing an `if` statement are predicate nodes where the conditional expression result of the `if` statement determines whether the statements referenced with an edge labeled with `T` (true) or `F` (false) are executed. We extend the control dependence edge labels by the execution order of the statements.

A node can be control dependent on multiple nodes. Consider the fallthrough of `case WRITING:` from Listing 1 Line 24 as an example. Nodes `[B9]`, `[B21]` and `[B19]` are control dependent on blocks `[B1]` and `[B17]`, i.e., the statements are executed if one of the conditions is true which is equivalent to the semantics of the fallthrough construct.

4.2.4 Traceability

Traceability throughout the whole normalization phase is maintained to support forward and backward navigation between the normalized program and the input program source code. This is especially required to support navigation from each element of the resulting state machine diagram to the associated source code locations.

The C preprocessor tracks the character locations during textual transformations. The tokens created during tokenization refer to the expanded locations. Each AST node created by the parser contains the locations and lengths of the represented tokens.

During the subsequent semantics-preserving transformations, newly created AST nodes are linked with the original AST nodes or nodes from previous normalization iterations. Changes in the control flow, e.g., swapping the *then* and *else* branches of an `if` statement to normalize the boolean condition of the `if` statement are stored in the AST node representing the `if` statement.

Each node of the control flow graph references an AST subtree representing a statement. Each node of the control dependence graph references a node of the control flow graph.

The resulting chain from control dependence graph node over control flow graph node over AST subtree over tokens to the text in the source code realizes the required fined-grained traceability. The mapping maintained by the C preprocessor during the textual transformations allows AST nodes and their tokens to be found in preprocessor macro definitions and the locations where they are referenced.

4.3 Pattern Matching

The pattern matching phase starts with the identification of the pattern declarations marked as entry points using the `@main` annotation (see Listing 4 Line 3). Each of these main pattern declarations is matched with each function of the input program using the respective control dependence graphs (see Figure 4 and Figure 3). For a pair consisting of a main pattern declaration and an input program function, a pattern instances tree (see Figure 5) with an instance of the main pattern declaration as root node is created. A pattern instance stores the values of local metavariables as well as the mapping of pattern CDG nodes to input program CDG nodes.

For each pattern declaration and function pair, a top-down analysis of the pattern control dependence graph is performed comparing the edges and nodes of the pattern with the corresponding elements of the in-

put program control dependence graph. Two nodes are compared by comparing the AST subtrees referenced by the nodes. For example, matching of `PStateCheck` with `Cyclic10ms` starts with matching pattern node `[B9]` (see Figure 4) with input program node `[B1]` (Figure 3). The statements referenced by the nodes are compared traversing the AST subtrees using a depth-first search. Two nodes match if all AST node pairs are equivalent. For variable metavariable references such as `stateVar`, the input program AST node must be a variable reference, e.g., `state`. The referenced variable declaration is stored in the metavariable. For constant metavariable references such as `stateConst`, the input program AST node must be a constant expression, e.g., a literal or reference to an enum member like `WRITING`. The successful matching of the two CDG nodes is stored in the pattern instance as mapping `[B9]`→ `[B1]`.

When the top-down analysis reaches pattern node `[B5]`, a new instance of the referenced pattern declaration `PStateCheck` is created and the matching is continued with the control dependence graph of the referenced pattern declaration. Pattern node `[B9]` is matched with input program node `[B17]`. Since the metavariable `stateVar` is global and has already matched in node `[B1]`, the variable referenced in node `[B17]` has to be the same as known from the previous match. Otherwise, the matching would not be successful, aborted and restarted for the next location in the program.

In contrast to `stateVar`, the metavariable `stateConst` is local and can be different for each pattern instance. In case of node `[B17]`, `state` is mapped to `WAITING_FOR_ACK`.

Input program nodes that are control dependent on multiple predicate nodes can be matched multiple times. For example, the input program node `[B9]` is matched twice. The `PGuard` pattern instance covering the second match refers to the node as `[B9′]`.

The `@level 0` annotation (see Listing 4 Line 3) lets the matcher start a matching process only at the top level of the input program. The `@coverage partialtoplevel` annotation configures the matcher so that a match of the pattern top level with a subset of the input program top level is sufficient for success. The pattern would therefore still match if statements were placed before or after the `switch` statement in Listing 1 Line 23.

4.4 State Machine Extraction

The state machine extraction phase uses the pattern instances tree and metavariable values provided by the matching phase to extract relevant information

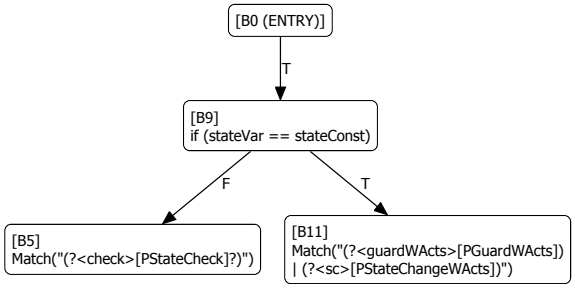

Figure 4: The *control dependence graph* of pattern declaration `PStateCheck` from Listing 4. Metavariable declarations are omitted since they are skipped during matching.

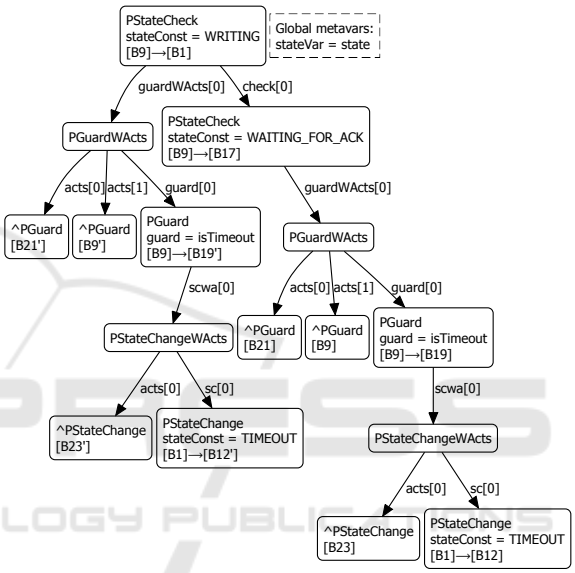

Figure 5: The *pattern instances tree* shows the instances of the pattern declarations from Listing 4 created during matching with the `Cyclic10ms` function from the running example.

from the input program. Control and data flow analysis results are also available for more complex extractions.

The extraction phase traverses the pattern instances tree (see Figure 5) using a depth-first search to build the state machine model. Following the `guardWActs[0]` edge of a `PStateCheck` pattern instance node, the current state can be inferred from the `stateConst` metavariable of the node. The value of the metavariable is the constant expression, i.e., the enum member, that represents the state. For the root node, the value is `WRITING`.

For each visited pattern instance of type `PGuard`, a choice pseudostate and a transition from the current state to the choice is created. An example for the pattern instance node that matches the input program node `[B19′]` is shown in Figure 6. The event

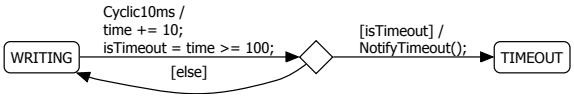

Figure 6: Transition mined from function `Cyclic10ms`.

of the transition is the function `Cyclic10ms` matched by the pattern declaration. The siblings of the `PGuard` pattern instance cover the input program nodes implementing the actions to be executed if the event is processed. The underlying statements are `time += 10;` and `isTimeout = time >= 100;`.

The `guard` expression `isTimeout` is stored for the eventual creation of the transition to the new state. In addition, a default transition with guard `[else]` is inserted that terminates the state transition if `isTimeout` does not evaluate to `true`. The aforementioned actions are still executed in this case.

Each time the traversal reaches a pattern instance of type `PStateChange`, a transition from the current state or the last choice pseudostate to the state identified by the `stateConst` metavariable is created. The previously stored guard expression is used as guard, if any. In this example, the guard expression `isTimeout` is used. The siblings of the pattern instance, i.e., the nodes referenced by the `acts[*]` edges of the parent node, are the actions executed with the newly created transition. The action introduced in Figure 6 is `NotifyTimeout();`.

The state machine models are iteratively extended by analyzing all pattern instances trees. Existing state machines are recognized via the state variable and existing states are recognized via the associated evaluated constant expression. The resulting state machine extracted from Listing 1 is shown in Figure 7.

The traceability allows to show pretty printed AST nodes of the AST created from the input program source code instead of the normalized code. Since each AST node contains the source code location, the model can be used, e.g., to replace the state machine implementation with a more sophisticated one. Formal verification would identify *TIMEOUT* as a deadlock state.

5 EVALUATION

We implemented a prototype and used online tutorials describing the implementation of state machines in C code using the nested `switch` statement technique to define the pattern and implement the extraction phase. We then applied the implementation to industrial code to evaluate our approach.

We used the first ten example code fragments from tutorials, which we could find online, to fine-tune the

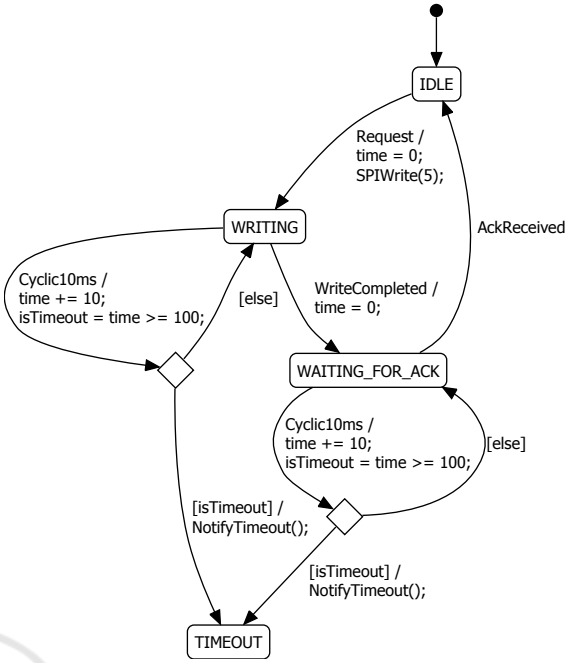

Figure 7: The complete state machine model mined from the C code shown in Listing 1.

pattern as well as the extraction implementation. We defined a single pattern that covers eight of these code fragments. The remaining two fragments used temporary variables and extracted the state handling in separate functions, i.e., the state check and transition are performed in different functions. These fragments could be covered using inlining and elimination of temporary variables in the normalization phase. However, these kind of normalizations have not been implemented, yet. The resulting pattern and extraction implementation support different variations, e.g., where the state is not checked at the top level but at an arbitrary level of nested guards. We further adapted the pattern to cover several corner cases. The resulting pattern has 118 lines of code.

We applied our state machine mining approach to handwritten industrial automotive C code from a center information display project. The project consists of 329 .c files containing 201,315 physical lines of code. We removed generated files and analyzed the remaining 286 handwritten .c files containing 147,333 physical lines of code. To define the *expected* results (ground truth), we started with a simplified pattern to extract state variable candidates possibly used to implement a state machine. A state variable candidate has to be a global or static local variable to be able to hold a state over multiple function calls. In addition, a state variable candidate has to be compared with a constant expression using an `if` or `switch` statement. At least one state change has to be directly or in-

directly control dependent on the stated comparison. The state change is expected to be an assignment of a constant expression to the state variable candidate.

The simplified pattern revealed 165 state variable candidates. We manually investigated all usages to remove candidates used as counters or where the state is fetched from external sources as well as pointers that reference different objects. We found 100 state machine implementations our approach should be able to extract.

We then applied our prototype to the code and verified the resulting state machine models for correctness and completeness. While typical analysis times range from several seconds to few minutes, 14 state machine implementations (see Table 1) located in three files could not be extracted since these files are too complex to be analyzed by our prototype. Four extracted state machine models could not be verified, because the resulting models contained so many transitions that the resulting diagrams could not be read. 74 state machine implementations could be completely extracted. Eight implementations could be partially extracted. The uncovered code fragments use temporary variables or are spread over multiple files which is not supported by our prototype, yet.

Table 1: Evaluation results applying our approach to industrial automotive embedded code.

Too Complex for Analysis	14
Too Complex for Verification	4
Successful	74
Successful With Limitations	8
Total	100

In summary: Our prototype could completely extract 74 % of the expected state machine implementations automatically. The results can be improved by introducing optimizing normalizations such as inlining and removal of temporary variables.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced a transformation- and pattern-based approach to extract UML state machine models from embedded code. Normalizing the pattern and input program covers several semantics-preserving variations. The approach can be used to extract state machines using different implementation techniques. We showed how to apply the approach to the nested `switch` statement technique. We performed an evaluation using a prototypical implementation and industrial automotive embedded code.

Future work will examine the extension of the pattern to extract complete statecharts (Harel, 1987) including orthogonal regions, hierarchical states and history. In addition, it has to be shown which implementation techniques besides the nested `switch` can be matched.

Further research is required to improve the semantic level of matching. This includes optimizing normalizations as known from compilers, e.g., inlining, dead code elimination, removal of temporary variables and loop unrolling. However, thorough optimization can break the required fine-grained traceability.

Complex embedded software is typically configurable and uses variability mechanisms using the C preprocessor. Our current approach requires to select a specific configuration before analysis. Future work will address variability-aware parsing (Kästner et al., 2011) to provide state machine models covering product lines.

The transformation- and pattern-based approach is not restricted to state machine mining. Future work will address other fields like the automatic evaluation of student programs. This will probably require to perform matching using the whole program dependence graph (Ferrante et al., 1987) to cover permutable statements.

Although the pattern is defined in the same language as the input program, the pattern definition is not trivial when it comes to recursion. For example, `switch` statements with an arbitrary number of `case` statements can be currently matched using a recursive pattern declaration with an `if` statement. We plan to simplify the pattern language, e.g., by allowing to use a `switch` statement with a repeatable `case`.

REFERENCES

Ammons, G., Bodík, R., and Larus, J. R. (2002). Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16.

Biermann, A. W. and Feldman, J. A. (1972). On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597.

Brunel, J., Doligez, D., Hansen, R. R., Lawall, J. L., and Muller, G. (2009). A foundation for flow-based program matching: Using temporal logic and model checking. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 114–126.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. (2000). Bandera: extracting finite-state models from java source

code. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 439–448.

Crew, R. F. (1997). Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349.

Fowler, M. (2013). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Grossman, D., Hicks, M., Jim, T., and Morrisett, G. (2005). Cyclone: A type-safe dialect of c. *C/C++ Users Journal*, 23(1):112–139.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274.

Jiresal, R., Makkapati, H., and Naik, R. (2011). Statechart extraction from code – an approach using static program analysis and heuristics based abstractions. In *Proc. of 2nd India Workshop on Reverse Engineering*.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. *SIGPLAN Not.*, 46(10):805–824.

Knor, R., Trausmuth, G., and Weidl, J. (1998). Reengineering c/c++ source code by transforming state machines. In *Development and Evolution of Software Architectures for Product Families*, pages 97–105.

Kung, D., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. (1994). On object state testing. In *Proceedings Eighteenth Annual International Computer Software and Applications Conference*, pages 222–227.

Ladd, D. A. and Ramming, J. C. (1995). A*: a language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901.

Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. (2002). Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compilier Construction*, pages 213–228.

Paul, S. and Prakash, A. (1994). A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475.

Prywes, N. and Rehmet, P. (1996). Recovery of software design, state-machines and specifications from source code. In *Proceedings 2nd IEEE International Conference on Engineering of Complex Computer Systems*, pages 279–288.

Ricca, F., Torchiano, M., Leotta, M., Tiso, A., Guerrini, G., and Reggio, G. (2018). On the impact of state-based model-driven development on maintainability: a family of experiments using unimod. *Empirical Software Engineering*, 23(3):1743–1790.

Said, W., Quante, J., and Koschke, R. (2018). Towards interactive mining of understandable state machine models from embedded software. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, pages 117–128.

Said, W., Quante, J., and Koschke, R. (2019). Do extracted state machine models help to understand embedded software? In *Proceedings of the 27th International Conference on Program Comprehension*, pages 191–196.

Samek, M. (2009). Practical uml statecharts in c/c++: Event-driven programming for embedded systems.

Sen, T. and Mall, R. (2016). Extracting finite state representation of java programs. *Software & Systems Modeling*, 15(2):497–511.

Shaohui Wang, Srinivasan Dwarakanathan, Oleg Sokolsky, and Insup Lee (2012). High-level model extraction via symbolic execution.

Somé, S. S. and Lethbridge, T. C. (2002). Enhancing program comprehension with recovered state models. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 85–93.

Thums, A. and Quante, J. (2012). Reengineering embedded automotive software. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pages 493–502.

van den Brand, M., Serebrenik, A., and van Zeeland, D. (2008). Extraction of state machines of legacy c code with cpp2xmi. In *Proceedings of the 7th Belgian-Netherlands Software Evolution Workshop*, pages 28–30.

Voelter, M., Kolb, B., Szabó, T., Ratiu, D., and van Deursen, A. (2019). Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, 18(1):585–630.

von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55.

Walkinshaw, N., Bogdanov, K., Ali, S., and Holcombe, M. (2008). Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability*, 18(2):99–121.

Walkinshaw, N. and Hall, M. (2016). Inferring computational state machine models from program executions. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 122–132.

Xiao, H., Sun, J., Liu, Y., Lin, S.-W., and Sun, C. (2013). Tzuyu: Learning stateful typestates. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 432–442.

Xie, T., Martin, E., and Yuan, H. (2006). Automatic extraction of abstract-object-state machines from unit-test executions. In *Proceedings of the 28th International Conference on Software Engineering*, pages 835–838.

Xu, S. and Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4):360–384.