Towards Interactive Mining of Understandable State Machine Models from Embedded Software

Wasim Said[1,2], Jochen Quante[1] and Rainer Koschke[2]
[1]*Robert Bosch GmbH, Corporate Research, Renningen, Germany*
[2]*University of Bremen, Germany*

Keywords: Model-driven Engineering, Program Comprehension, Software Analysis, Reverse Engineering, Model Mining.

Abstract: State machines are an established formalism for specifying the behavior of a software component. Unfortunately, such design models often do not exist at all, especially for legacy code, or they are lost or not kept up to date during software evolution – although they would be very helpful for program comprehension. Therefore, it is desirable to extract state machine models from code and also from legacy models. The few existing approaches for that – when applied to real-world systems written in C – deliver models that are too complex for being comprehensible to humans. This is mainly because C functions are typically much longer than object oriented methods, for which these approaches were originally intended.
In this paper, we propose and investigate different measures to reduce the complexity of such mined models to an understandable degree. Since the code alone does not contain all required information for abstraction, user interaction is essential. Also, different users will be interested in different aspects of the code. Therefore, we introduce several possibilities for influencing the state machine extraction process, such as providing additional constraints for reducing the state space. We show the effectiveness of these interactions in several case studies. The combination of these interactions gives the user a rich set of possibilities for exploring the functionality of the software.

1 INTRODUCTION

Model mining is the extraction of behavioral and/or structural models from existing software systems. For example, a lot of tools exist for extracting UML class diagrams from Java code. These tools extract explicit structural information from the code and bring them into a different (graphical) form. The resulting models are quite helpful for getting an overview and understanding at a higher (design) level. We are interested in extracting more implicit information that is not explicitly visible in the code. An example for that is the extraction of a state machine model from C code when no explicit state machine pattern can be found in the code.

Having such higher-level models can help developers in several ways. Firstly, it helps in program comprehension: Developers often try to manually reconstruct such models during maintenance tasks (Roehm et al., 2012). This is a quite time-consuming activity: Program comprehension makes up for 40%-50% of total software life cycle effort (Fjeldstad and Hamlen, 1984). Secondly, model mining can also be a great support for migration towards model-based software development. Since companies that want to switch to model-based development usually do not start from scratch, but already have a large code base, extraction of models from legacy code would be quite helpful. For example, model-based development tools such as ASCET[1] or Matlab Simulink[2] have been introduced at almost all automotive companies. Their block diagram models provide an adequate basis for implementation of control systems and thus help to save cost and time (Broy et al., 2013). However, only new functions are usually implemented based on these models. The existing code base already realizes a huge amount of functionality and cannot be easily replaced by corresponding models. Model mining can support this transformation and even make it economically worthwhile. Thirdly, even when developing exclusively based on models, developers do not always use the best-suited

[1]https://www.etas.com/en/products/ascet_software_products.php
[2]https://www.mathworks.com/products/simulink.html

Said, W., Quante, J. and Koschke, R.
Towards Interactive Mining of Understandable State Machine Models from Embedded Software.
DOI: 10.5220/0006593501170128
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 117-128
ISBN: 978-989-758-283-7

models for a given aspect. For example, control logic is sometimes modelled as a block diagram – which makes the aspect "control logic" very hard to understand. The extraction of the behavior of a function with respect to control logic in form of a state machine is then very desirable and helpful for experts – for both legacy code and models.

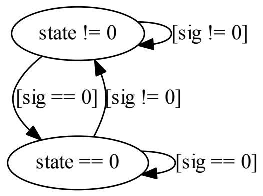


Figure 1: Extracted state machine for example function.

As a simple example for state machine extraction, consider the following C code:

```
int biEdge(int sig) {
  static int state = 0;
  if ((sig && !state) || (!sig && state)) {
    state = sig;
    return 1;
  }
  return 0;
}
```

The static variable `state` is the only variable that holds state, and it is used in the conditions `state` and `!state`. Consequently, a state machine representation of this code would distinguish between these two states. Figure 1 shows the resulting state machine. Although simple in this example, this kind of model extraction can be very laborious. This is the case when more state variables and conditions are involved, or when they are interleaved with other code that is not relevant for state in a larger function. Therefore, automation of this process is highly desired. However, fully-automatic model extraction from real world systems results in information on the wrong level: The automatically mined models are typically too detailed and low-level, because code alone does not contain all the necessary information. For example, there is no information about which details are important and which are not. Also, a tool is not capable of introducing abstractions that a human would immediately come up with. Consequently, effective model mining requires a combination of automation and interaction. The automated part must be able to utilize expert knowledge and feedback, so that it is capable of extracting highly useful models with low manual effort.

Our approach for an interactive state machine extraction process is as follows: The developer has some function which he wants to understand with respect to sequencing logic. He then starts the model mining process on this function and gets an initial result, which usually is far too complex to be understandable. Now he has several options to reduce the complexity of this state machine. For example, he can select a subset of state variables for the state machine model or join certain states. Alternatively, he can provide additional constraints – for example, set variables to certain values, or limit their ranges. The mining tool then returns a different state machine model – one that considers this additional information. This interactive process can go through multiple iterations with changed user input. In any case, the resulting model will be more abstract and potentially closer to the domain and thus less complex. When interactively working with the model extraction process this way, the developer can extract understandable state machines for different scenarios which in combination can provide a complete picture of the relevant functionality.

Our contributions in this paper are the following:

- An adaption of an existing (Kung et al., 1994; Sen and Mall, 2016) state machine mining approach for procedural code.
- Several automatic techniques for simplification of the extracted models.
- A set of interactive state machine mining extensions for reducing the state space of extracted models.
- Demonstration of the effectiveness of these measures for making the resulting models understandable.
- First feedback from experts about usefulness of the resulting models.

The rest of the paper is organized as follows. The required background about static analysis techniques and the approaches of Kung and Sen are introduced in Section 2. Our adaptations and optimizations of Kung/Sen's approach are presented in Section 3. Section 4 explains our different techniques for interactive exploration of state machines. The case studies are presented in Section 5. In Section 6, we discuss the approach and results. An overview of related work is presented in Section 7, and Section 8 concludes.

2 BACKGROUND

In this section, we present an overview of analysis techniques that are used in our approach.

2.1 Symbolic Execution

One widely used and effective static analysis technique is symbolic execution. The idea behind it is the execution of a program with symbolic values instead of concrete values. Traditional symbolic execution performs path enumeration for test case generation with full path coverage (King, 1976). When symbolic execution comes to a branching point, all branches are explored in a depth-first search manner. The valid conditions at each branching point are collected and stored with the corresponding path. The technique backtracks when it comes to a point where the path condition is not satisfiable, i. e., when the path becomes infeasible. For every possible path through the program, this technique delivers the path condition (*PC*) and the output variable values as symbolic expressions over the input variables. All possible distinct behavior of the function is therefore described with just a single control condition – the path condition. As stateful behavior means different behavior depending on state, this information provides a good basis for deriving state machines: States are contained in path conditions, and each path corresponds to a certain transition.

2.2 Kung/Sen Approach to State Machine Mining

Kung et al. (Kung et al., 1994) describe how symbolic execution can be used to derive a state machine representation from a given C++ class. Sen and Mall (Sen and Mall, 2016) add some improvements and work on Java classes. To explain the approach, let us first define some terms that were introduced by Kung. An *atomic condition* is either a boolean symbolic variable such as a or a relational operator over two symbolic expressions such as $(a+b<5)$. Connecting atomic conditions by conjunction, disjunction or negation creates a *compound condition*, such as $((a+b>5)\wedge d)$. A *conditional literal* is an atomic condition or the negation of an atomic condition.

The approach works in three steps: First, symbolic execution is performed. It generates the path conditions, pairs of updated member variables during the execution of each path and the returned symbolic expression (if any). In a second step, states are generated from that, and in the third step, transitions are determined.

Extraction of States

Kung's state generation approach examines all path conditions and considers only conditional literals that contain exactly one member variable. For each such conditional literal, it partitions the variable's domain into intervals. The cross product of the intervals of all member variables then generates the states and their invariants.

For example, imagine that x is an integer variable and occurs in the conditional literals $(x>0)$ and $(x=3)$. The relevant intervals of x, according to Kung et al., are $(-\infty,0]$, $(0,3)$, $[3,3]$ and $(3,\infty)$. This approach works only on conditions that contain exactly one member variable. For conditions with more than one variable such as $(a<b)$ and $(a+b+c>5)$, computing the intervals is not possible. These conditions are ignored by Kung's approach.

Sen and Mall address this limitation and extract states from path conditions in a different way. They directly compute the partitions from the original conditions. Then, they use a solver to find out which of them are satisfiable; those become the state candidates. For the above example, the extracted partitions for Sen's approach are: $((x>0)\wedge(x=3))$, $((x>0)\wedge(x\neq 3))$, $((x\leq 0)\wedge(x\neq 3))$ and $((x\leq 0)\wedge(x=3))$ (the latter is not satisfiable). It is obviously also possible to create such partitions for conditional literals with more than one variable.

Sen and Mall distinguish between different types of conditional literals, depending on the kind of different variables involved (in an object-oriented program):

- *Member dependent literal (MDL)*: a conditional literal in which only member variables of the class appear.
- *Parameter dependent literal (PDL)*: a conditional literal in which only parameters of a class method appear.
- *Mixed literal (MXL)*: a conditional literal which is neither MDL nor PDL.

Although we do not have member variables in C code and ASCET models, we stick to these definitions. For us, *members* correspond to *state candidates* (see Section 3). Because only MXLs and MDLs are practically relevant for state machine extraction, we also use the term *pure literal* for MDLs. Only MDLs are considered in Sen's approach, and only MDLs with a single variable are considered by Kung. We leave the choice of including MXLs in the states to the user, which is discussed later in this paper.

The described state space generation process may generate a lot of states that can never be reached. By providing (or extracting from initialization) a set of potential start states, all states that are not reachable from them can be eliminated, and all unreachable transitions can be removed as well. Kung et al. already considered this in the construction of their state space, but it may as well be performed as a post-processing step, as we have done in our approach.

Extraction of Transitions

For transition generation, it is then examined for each path from which state it can possibly start (*pre state* S_{pre}). This is again done by using a constraint solver. It checks for which state candidates $Inv_{S_{pre}} \wedge PC$ is satisfiable, Inv_S is the state invariant of state S. The *post state* S_{post} is determined in different ways. Kung et al. use the condition $Inv_{S_{pre}} \wedge PC \models Inv_{S_{post}}(E)$, which means that the target state condition $Inv_{S_{post}}$ on the resulting symbolic values of the path (E) must be satisfied in *all* cases. Sen just requires that $PC \wedge Inv_{S_{pre}}$ is satisfiable, and that $PC \wedge Inv_{S_{post}}(E)$ is satisfiable independently. In our approach, we do not check these two conditions independently, but we combine them so that $PC \wedge Inv_{S_{pre}} \wedge Inv_{S_{post}}(E)$ must be fulfilled.

2.3 Concolic Testing

Concolic testing is a more recent technique for path enumeration (Godefroid et al., 2005). The main difference to symbolic execution is that it does not explore paths by forking at the branching nodes, but explores paths sequentially. It executes each path with concrete values and in parallel collects the symbolic path condition. The input for the next execution is inferred from the path conditions of the previous test cases. This is done using an SMT solver. By negating the disjunction of all path conditions that were so far encountered, it generates new values for the input variables that lead to execution of another path that has not been covered yet.

For instance, for a branching condition $(x > y)$, the concrete values could be $x = 3$ and $y = 1$ which satisfies the constraint and causes the execution of the true branch. To execute the false branch, concolic testing tries to generate values that do not satisfy the constraint. This can be done by negating the constraint $(x > y)$ and using a constraint solver to generate values that satisfy $(x \leq y)$.

Concolic testing has some advantages over symbolic execution. The key limitation of symbolic execution is constraint solving. Some constraints, such as non linear ones, cannot be handled by solvers. In concolic testing, complex constraints can be solved with concrete values in certain cases. See (Hoffmann et al., 2016) for a detailed discussion of differences between symbolic execution and concolic testing.

3 ADAPTATIONS

In the following, we present three adaptations of Sen and Kung's approaches for our setting. Firstly, our systems are written in C or ASCET, so we need a different way to identify state variables. Secondly, we use concolic testing instead of symbolic execution. Thirdly, we discuss how to deal with mixed literals.

3.1 Adaptation for C Code

The approaches by Kung and Sen both work on object-oriented software. All member variables that influence any conditions are potential state candidates there. In our case, we have to deal with C code or ASCET models, where there are no natural member variables. Therefore, we have to come up with additional heuristics to determine the relevant state candidate variables: The variable must keep its value over multiple invocations, which means it has to be global or static, and the previously written value has to be used. Also, it has to influence a control decision through control or data dependencies. Otherwise, it does not hold state.

3.2 Use of Concolic Testing

Kung and Sen depend on symbolic execution to perform path enumeration and generate test cases. We use concolic testing instead. Despite the advantages as discussed in Section 2.3, concolic testing also better supports our interactive extensions, such as adding user constraints or reducing state variable range. These interactions are implemented by feeding additional constraints into the state machine mining process. Concolic testing solves constraints incrementally with every newly encountered path. It deals with additional constraints from a user the same way, which is quite efficient. Consequently, concolic testing is the more adequate choice for this use case.

3.3 Mixed Literals

Reduction to the relevant state variables may be achieved by investigating the conditions on these state variable candidates: if all conditional literals in which a state variable occurs are MXLs, the state would always be changed according to the input variable's value. For example, consider the following function:

```
int changed(int x) {
  static int old = 0;
  if (x != old) {
    old = x;
    return 1;
  }
  return 0;
}
```

The resulting states would be $x = old$ and $x \neq old$. However, the state after a call to this function always is $x = old$, since `x` is assigned to `old`. Therefore, it does not make much sense to consider $x = old$ and $x \neq old$ as states in this case. Sen et al. *always* ignore such mixed variables.

In other cases, these MXLs may carry important information for the user – they may be helpful for program understanding. As an example for that, the state machine model in Figure 2 was extracted from the test function MON (cf. Section 5) and contains states with the variables `LmpOn` and `TmMAX`. `LmpOn` is a state variable, but it occurs in mixed literals with the input variable `TmMAX`. The state machine model was reduced to state variable `Timer`, which appears only in MDLs, along with `LmpOn`, which occurs only in MXLs. The information about how states can change according to the input variable shows an interesting fact: Once `LmpOn` has become lower or equal to `TmMAX`, it can never raise to a value higher than the value of `TmMAX` again.

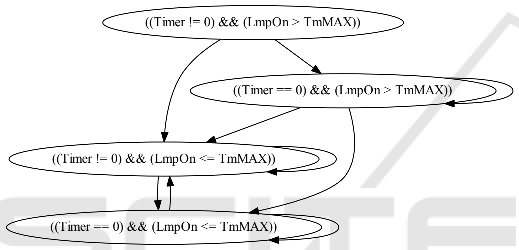

((Timer != 0) && (LmpOn > TmMAX))
((Timer == 0) && (LmpOn > TmMAX))
((Timer != 0) && (LmpOn <= TmMAX))
((Timer == 0) && (LmpOn <= TmMAX))

Figure 2: Example for the potential importance of mixed literals (function MON).

In Section 5.1, we will demonstrate that ignoring mixed variables can effectively make state machine models more understandable. However, because they can carry important information in some cases such as the above example, we do not ignore them in our approach. Instead, we let the expert decide which input variables remain and which do not. This is done by presenting a list of all relevant input variables along with their relevant conditions and letting the user choose.

4 INTERACTIVE EXPLORATION

One big problem of the approaches of Kung or Sen is the state space explosion. Their approaches quickly generate hundreds of states and transitions that a) are impossible to understand and b) require a long time to extract. Our goal is to extract state machines that help in program understanding. To reduce the state machines' complexity to a degree that makes them understandable for humans, we have to come up with additional measures. In this section, we introduce a number of interactive approaches to complexity reduction. In the evaluation section, we will then investigate their effectiveness. An overview of the extraction process and the possible user interactions is depicted in Figure 3. The different interactions are presented in detail in the next sections.

4.1 State Variable Subset

The first possibility to reduce the state space is to manually restrict the relevant state variables to a subset. This should lead to simplified state machines that can be understood more easily. A comprehensive group of such simplified state machines may still provide a full picture of a function's behavior: It does not make sense to describe the entire functionality of a larger function in a single diagram, but it is usually more useful to focus on certain aspects at once – and abstract away the rest.

In this step, the user is presented the full list of identified *state variables* along with their relevant values or value ranges. She can then select those variables that are potentially interesting. Table 1 shows an example how this information is presented. This interaction corresponds to selecting or deselecting an entire row in this table.

It may be difficult for the expert to decide which variables to include and which to ignore just based on the variables' names. Therefore, we use program slicing (Weiser, 1981) to calculate the state variables' effect size, i. e., how much code (the number of PDG nodes) is dependent on this state variable. The variables are sorted by descending effect size, starting with the most influential one. This can be used by the expert as an indication of relevance of each variable.

Table 1: Information about the first three state variables, their effect size and relevant values/ranges for function SPD.

State variable	Effect	Values/Partitions
Ctr	0.60	=0 | =1 | =2 | =3 | other
SPD_mode	0.30	=4 | ≠4
Sync	0.27	true | false

4.2 Joining States / Reducing State Variable Alternatives

Reducing state variable alternatives is another option to reduce the state space. For example, there may be conditions "*speed* in $(0, 50)$" and "*speed* in $[50, \infty)$", but the user only wants to distinguish between $(speed = 0)$ and $(speed > 0)$. The two ranges that satisfy the condition $(speed > 0)$ are then unified, leading to a reduced state space as well. As described in the previous section, the list of relevant value or range alternatives for each state variable is presented to the user. The user can decide to either ignore or merge certain alternatives for state machine model extraction based on this information. This corresponds to merging different alternatives in column *Values/Partitions* in Table 1. The user can also do that by adding the corresponding constraints, as shown in the next section.

4.3 Constraints

With a reduced number of paths, the extracted state machines also become less complex. For example, if certain variables are set to a fixed value, this may reduce the number of paths and thus also the number of possible states and transitions, and it may reduce the complexity of state invariants and transition conditions. Another possibility is to generate paths under more general constraints, such as $x = y + 1$. Which constraints make sense largely depends on the system under consideration.

For instance, the expert may be interested in investigating the scenario in which the software is in a certain mode (e. g., “driving”) and the speed is lower than 20 km/h. Here, *mode* and *speed* are state variables. The expert can also define constraints on non-state variables, which will also reduce the number of states and transitions.

For now, we leave it to the expert to decide which values to restrict or which additional conditions to consider during the extraction of the state machine. For example, the expert may want to investigate a scenario where all possible error conditions are excluded. The decision-relevant conditions on state variables that are automatically extracted may be helpful to find out relevant constraints. Automated identification of meaningful constraints is a topic for future research.

This interaction is supported by the list of variables that appear in the program. The user can select variables and their values or ranges of values in the same way. Additionally, the user can write her own constraints in the form of expressions on these variables (plus literals). The constraints can also contain any combination of conjunction, disjunction and negation. This gives a very high flexibility to the user’s choice of constraints that she would like to impose on the extraction process and allows her to investigate arbitrary scenarios.

4.4 Interactive Process

Figure 3 shows the interactive state machine extraction process. The analysis knowledge base contains information about all variables, states and transitions

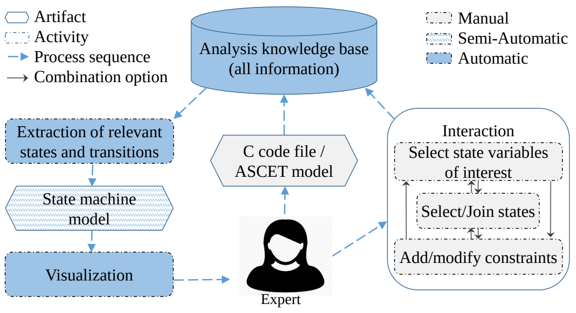


Figure 3: Interactive state machine extraction process.

which are automatically extracted from the code. It also contains the information that is added by the user, such as constraints or state subsets. The approach can leverage all this information for extraction, and the user can iteratively perform any interactions from above. This will result in a reduced state machine model according to the user’s needs. With each user interaction, the extracted model is automatically updated. The process iterates until the user gets the model that she needs for the task at hand. The different interaction possibilities can be used not only for general simplification of state machine models, but also for specific purposes like investigating what-if scenarios, e. g., by setting some variables to certain values. These interactions support the extraction of different projections of the detailed state machine model. However, they also support reduction of the original model to a meaningful and comprehensive model for migration towards model based development.

5 CASE STUDY

In order to verify the effectiveness of our approach, we implemented it in a prototype and used that for performing several case studies. The implementation is based on our software analysis framework (Quante, 2016), which includes frontends for C and ASCET, control/data flow analysis, concolic testing and an interface to the Z3 SMT solver. On top of that, we implemented Kung/Sen’s approach for state machine mining with the adaptations as described in Section 3. Furthermore, we added all interactive extensions that are presented in Section 4 of this paper.

The studies were conducted using four C code and five ASCET (A) real world functions from two different automotive software systems. Table 2 summarizes their characteristics. Lg is language, LOC denotes lines of code (ASCET: ESDL code), MCC is McCabe’s cyclomatic complexity, NPATH is Nejmeh’s static approximation of the number of paths (Nejmeh, 1988), and #var is the total number of variables. These functions were nominated for state machine model extraction by developers because they found them to be hard to understand with respect to control logic.

Table 2: Characteristics of subject functions.

	Lg.	LOC	MCC	NPATH	#var.
SPD	C	807	52	34,560	25
MON	C	75	18	41,472	12
AVG	C	91	20	262,145	16
VDA	C	410	54	863,115	39
PRK	A	104	30	284,756	33
RSK	A	169	27	147,480	56
VPR	A	110	17	10,368	17
SSD	A	58	16	2,816	13
PPR	A	49	16	86	10

The C functions are part of an engine control software. The SPD function is responsible for switching between different possible input signals when certain conditions apply. It partly contains an explicit state machine with a state variable and switch-case construct, but it contains additional distributed logic that makes it very complicated. The MON function contains a timer and checks whether input signals change in a certain way within a specified time. The AVG function calculates several kinds of average signal values according to a clock signal. VDA is a function that only becomes active after a certain sequence of events has occurred. The ASCET models are taken from another automotive control unit.

The approach as described above – excluding interactive measures, but including our other optimizations and both pure and mixed literals – results in state machines as shown in Table 3. Obviously, the models contain a large number of states and transitions and thus require a long time for understanding (Miranda et al., 2005). This means the majority of these models is too complex for human comprehension.

Table 3: Characteristics of resulting state machines.

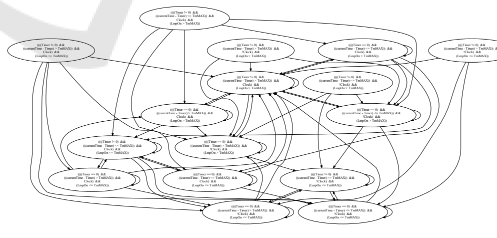
	#state var.	#states	#transitions
SPD	6	720	1,652
MON	3	16	64
AVG	6	64	184
VDA	10	6,144	19,127
PRK	4	36	112
RSK	5	48	170
VPR	3	1,157	5,904
SSD	2	32	160
PPR	3	8	16

In the following, we investigate and demonstrate the effect of different user interactions separately. However, the use of any combination of these scenarios is fully supported and desired. We first concentrate on the structure and complexity of the resulting state machines regarding the number of states and transitions. Therefore, we do not show transition conditions in the figures.

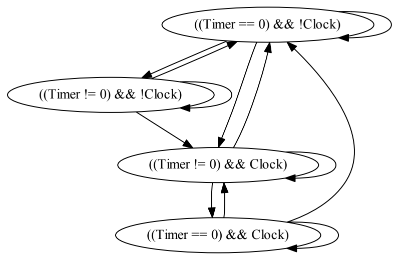
5.1 Mixed Literals

In order to demonstrate the effect of ignoring mixed literals (MXLs) for state generation, we compare the number of extracted states and transitions from all state variables of the program, first including MXLs and MDLs, and then only considering MDLs.

Figure 4 depicts the extracted state machine model from function MON. There are three state variables in the code used to generate the state machine model with and without mixed literals. The model has 16 states and 64 transitions when both pure (MDL) and mixed literals (MXLs) are used. The structure of the graph alone is obviously too complex to be understandable. Using only pure literals reduces the number of states to four and the number of transitions to 12. This is because the conditional literals on the first state variable all depend on input variables, i. e., they are all MXLs. One of the two conditions on the second variable contains input variables. Therefore, removing these mixed literals results in a state machine model with only two state variables. The number of transitions is consequently reduced, because all transitions from and to the removed states are also deleted. The resulting state machine is obviously much less complex, but it still gives a good picture of which sequences of operation are supported



(a) MDLs and MXLs



(b) only MDLs

Figure 4: The effect of using only pure literal conditions in contrast to using pure and mixed ones (function MON).

by the software, and which are not. For example, the state `((Timer == 0) && Clock)` can only be reached through `((Timer != 0) && Clock)`.

The effect of using or not using mixed literals on the other functions is summarized in Table 4. The results are presented as a percentage of the original number of states and transitions (with pure and mixed conditions and without any user constraint). The first column shows the number of states (#S) and transitions (#T) with both pure and mixed conditional literals, and the second column shows the same information for only pure conditions (as the remaining share of states and transitions). Function VPR does not have any pure conditions: All the conditional literals depend on parameters. This emphasizes again that these conditions cannot always be ignored (cf. Section 3.3). On the other hand, function PRK has only pure conditions.

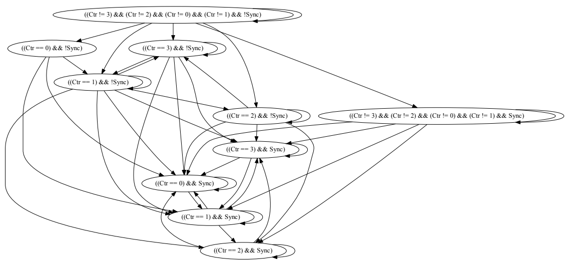
Table 4: Model reduction: The effect of using only pure conditions and user constraints on the number of states and transitions.

	Pure + Mixed		only Pure		UserCon Pure + Mixed		UserCon only Pure
	#S	#T	%S	%T	#C	%T	%T
SPD	720	1652	1 %	2 %	12	1 %	0,5 %
MON	16	64	25 %	18 %	6	81 %	15 %
AVG	64	184	25 %	34 %	4	69 %	17 %
VDA	6144	19127	2 %	2 %	4	45 %	1 %
PRK	36	112	100 %	100 %	5	49 %	49 %
RSK	48	170	4 %	1 %	3	56 %	0,5 %
VPR	1157	5904	0 %	0 %	5	35 %	0 %
SSD	32	160	12 %	10 %	4	63 %	6 %
PPR	8	16	25 %	25 %	2	75 %	18 %

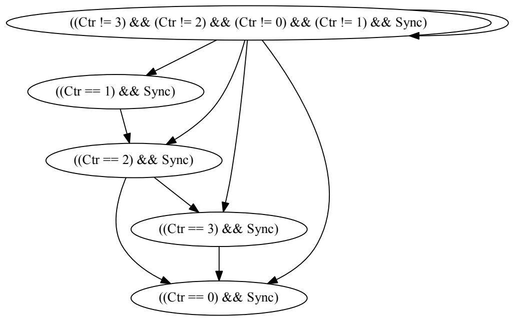
5.2 User Constraints

As mentioned in Section 4.3, the search space (i. e., number of paths) can be reduced by generating paths under additional constraints. The user can add such constraints. Because she may have no idea about the analyzed system, we first show her a list of not only state variables, but *all* variables that have any effect on any branch condition. This list includes the conditional literals for each state variable. The user can select the variables and either set certain variables to fixed values or enter constraints on them. We then additionally feed these constraints into the state machine extraction process – simply by using them as additional constraints in concolic testing.

Figure 5 illustrates the state machine model of SPD. In this scenario, we consider only pure conditions. SPD has six state variables, which end up in 10 different states and 35 transitions between them. In addition, SPD includes 19 non-state variables. In this experiment, we have randomly selected 12 of these non-state variables and set them to specific values or range of values. The resulting model has the same



(a) No additional constraints.



(b) With extra user constraints.

Figure 5: The effect of additional user constraints (SPD).

number of states (10), while the number of transitions has been reduced to nine[3]. The reason is that the search space becomes much smaller than the one without extra conditions from the user.

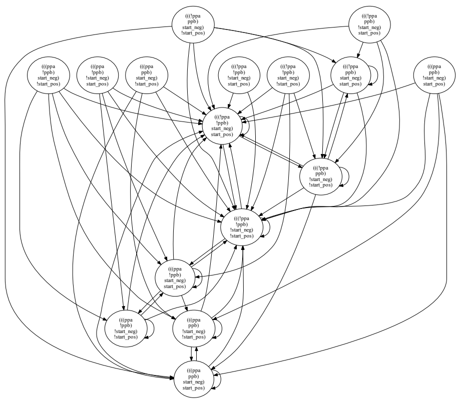
Table 4 shows how user constraints reduce the model from the other eight test cases. The third column gives information about the used number of user constraints (#C) and the share of remaining transitions (%T) with pure and mixed conditions. The last column shows the same information – with the same user constraints – but with only pure conditions. The used user constraints were only applied on non state variables in this experiment. Therefore we show only the effect on the number of transitions (%T). Constraints on state variables are discussed in the next section.

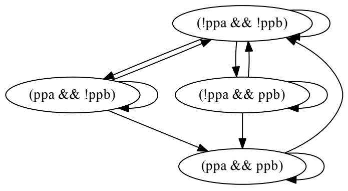
5.3 State Variable Subset

Instead of generating a state machine model with all state variables in a program, the user can also select a subset of state variables that she is interested in. This obviously leads to a state machine model with a lower number of states. For this interaction, the user can choose any combination of state variables, which results in a wide variety of alternatives. Therefore, we show only one example from the large number of experiments that we have done.

Take the AVG function as an example. It contains six state variables. All these variables (with only

[3]In the figure, we show only five states, because the other five states are not connected by any transitions.


(a) All state variables.



(b) Only two state variables.

Figure 6: The effect of reducing the set of state variables (AVG).

pure conditions) generate a model with 16 states and 64 transitions. We applied the effect size heuristic to get a hint which state variables would be most interesting. Of the six state variables, the 2nd and 3rd one (according to the effect size) turned out to be the most interesting ones. This was done by interactive exploration of different subsets, starting with the ones with the highest rating. Selecting only these two variables reduces the number of states to 4 and transitions to 11. The models with all and with only these two variables are shown in Figure 6. The result is a significant complexity reduction and models that concentrate on certain aspects of the overall state.

5.4 Reduced State Variable Range

For the examples so far, state variables consider all possible values in the program. However, our approach allows the expert to restrict values or ranges of values for state variables.

Please recall Figure 5(a) in Section 5.2. That model represents *all* possible states of *Ctr* and *Sync* variables. The relevant values for variable *Ctr* are $\{0,1,2,3\}$. When the user is interested in only two values like $\{0,1\}$, she can select these values and gets the corresponding state machine model, which is depicted in Figure 7. Obviously, the state machine becomes much simpler. The resulting state machine is a compact model of a quite specific aspect of the function. Similar to the previous section, we could only

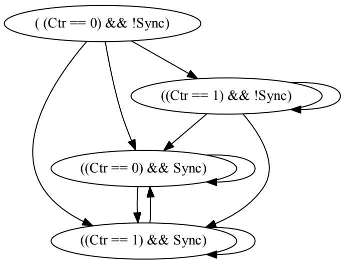


Figure 7: The effect of reducing state variables' range (SPD).

show one example due to the high number of interactive options.

All interaction possibilities can be used independently or in combination with other interactions. This depends on the user's goal and her previous knowledge of the system (not necessarily software). For example, when the user knows the states of interest, she can select the right state variables and their values at once. When the system is new to her, she can use the different interactions in an explorative way to gain knowledge about the software.

5.5 Experts Feedback

The main goal of our work is to extract understandable and useful state machine models from code. How helpful the extracted models are can only be determined by humans who work with the code. For this reason, we conducted interviews with three developers, whose knowledge about the behavior of our subject functions ranges from expert (function responsible) to people with no idea about the subject system. We presented different extracted state machine models with randomly selected states and constraints to each developer. Then we asked the developers whether these models were helpful or interesting to them, and what they can read out of them.

The feedback from the expert was quite interesting. He thought that some transitions in the extracted models would not exist in the code. However, when he reviewed the code, it turned out that the transitions do exist. Therefore, we can say that these models can also help in validation, verification and debugging. One other interesting reaction was that he immediately came up with ideas which states would be interesting and which constraints he would like to try. Feeding the relevant information and constraints to the process does not seem to be a problem to him.

The developer who had at least an idea about the functionality of the subject system stated that the extracted models give him valuable insights into the function, which would be a lot of work to get from code: "The model at this level of detail is already very helpful, since I can validate the possible states and transitions with my expectations. The proposed approach of interactive scenarios and projections should help to make the complexity more manageable."

Finally, the developer without any knowledge was asked to draw some conclusions about the models. It was interesting that he was able to understand the models and determine the main ideas quickly. He determined which states exist, what they mean and how they can be reached. When we compared these conclusions with the available information from the function authors, they were correct. For example, he was able to come to the fact in Figure 2, that `LmpOn` can never raise to a value higher than the value of `TmMAX` once it becomes lower or equal to `TmMAX`.

In summary, the feedback shows that the extracted models can be helpful to developers with different levels of system knowledge. This is true even for the expert who has worked with the function for a longer time. These early results are quite promising and encourage further research and refinement of our interactive approach.

6 DISCUSSION

The case studies in the previous section showed that each individual interaction with the state machine extraction process has the potential to lead to less complex but still useful state machines. The combination of these techniques offers even more possibilities: The user can control focus and complexity of the extracted models according to the question at hand. Given these results, the applicability of such a process on complex real-world systems seems to be feasible and effective.

The first feedback from experts is also quite promising: They seem to have good ideas about which scenarios they want to investigate, and also which constraints make most sense. So both technically and from user perspective, our interactive approach appears feasible. However, we are aware that our case study is only a first step towards proving the effectiveness of our interactive approach.

We concentrated on states and transition structure so far and ignored transition conditions. However, the state/transition structure alone was already considered quite useful by our developers. Transition conditions quickly become quite complex and unreadable. Making transition conditions comprehensible for users is part of our future work.

Since this state machine mining approach intensively uses an SMT solver – for path enumeration, for determining feasible states, and for finding out pre and post states for each path –, the computational cost is very high. For an interactive approach, short response times are required. The interactive approach also addresses this issue and strongly affects the required time for the model mining process: By providing additional input, limiting ranges or giving additional constraints, the number of paths and states is largely reduced. The number of solver calls is in $O(P * S^2)$ with P number of paths and S number of states. Therefore, specially reducing the number of states also drastically reduces the required computation time. For example, extracting the state machine model with all state variables from the function AVG takes 16 seconds. When extracting this model with only three variables, it takes only 2.5 seconds. We have actually observed a decrease of computation time when using interactions in all cases.

Our approach can meet the needs of different users with different backgrounds about the system. For example, domain experts or users with good knowledge of the system can directly select the states or constraints of interest. Users with no idea about the system get a ranking of all state variables according to their effect size. We do not yet provide such information about the relevant constraints that affect specific states in the code, but such information could also be quite useful for this usage scenario in the future. In our study, we have randomly selected some constraints and set them to certain values, and the result was very good. When the user had some ideas about which constraints have the highest effect on states, the result would be even better. Therefore, mining of relevant constraints that affect specific states could be helpful.

Our case studies and first interviews with experts suggest that this approach can not only be helpful for program comprehension and migration towards models, but also for validation, verification and debugging tasks. These different usage scenarios make the approach very attractive. We will continue our work towards further improvement, specially with respect to the challenges mentioned above.

7 RELATED WORK

Most of the work on extraction of state machines deals with API protocols, i. e., the allowed sequences of API calls. The seminal work on this topic was done using dynamic analysis by Ammons et al. (Ammons et al., 2002). Similar protocol extraction based on static analysis was first introduced by Eisenbarth et al. (Eisenbarth et al., 2005). Little work is done to extract state machines that describe the behavior of an application. We focus on these approaches in this section.

Bandera (Corbett et al., 2000) automatically extracts state machines from the source code of Java programs for model checking. The generated models are in the input language of one of the used verification tools. The resulting models with thousands of states are not a problem for this use case – but for human comprehension, they are. In the publication of Prywes et al. (Prywes and Rehmet, 1996), the user selects the desired software components like functions or operations in the process of generating state machines. However, the extracted state machine models from both of these works are on a very low level, which makes them unusable for human understanding or model based development. Prywes also concluded that the extraction process must be human-guided.

Xie et al. (Xie et al., 2006) extract state machines from object-oriented code based on tests. They reduce their size by using additional information, such as test coverage information, observer functions, and state slicing. The latter corresponds to our approach of ignoring certain state variables. However, dynamic information, which must be representative to be usable in this approach, is usually unavailable for real-time systems.

Some approaches can only be used for specific cases like in the work of Abadi et al. (Abadi and Feldman, 2012). They extract state charts from specific patterns generated from code generators, so it cannot be applied for hand-written software. Somé et al. (Somé and Lethbridge, 2002) assume that the state is represented by only one variable, which restricts the implementation of this approach to systems with obvious state machines.

Walkinshaw et al. (Walkinshaw et al., 2008) assume that all states were identified manually in terms of logical conditions on program variables. Then, the approach uses symbolic execution to extract transitions between these states. It identifies the paths in the source code that correspond to the extracted transitions. This approach also assumes that all transitions are mapped to functions. Our approach does not have these assumptions. In another paper (Walkinshaw and Hall, 2016), Walkinshaw uses genetic programming to infer computations for state machines. However, this approach is in a very early and experimental stage and not applicable to real-world software.

Wang et al. (Wang et al., 2012) extract high level state machines of GUI-driven software for cell phones and PDAs, where screens are modeled as states and the calls of UI components as transitions. This approach is quite specific to the used GUI framework and does not serve our purpose of extracting high level state machines from real world software intensive systems.

An approach to extract state charts from code was proposed by Jiresal et al. (Jiresal et al., 2011). They use static data flow analysis and heuristics based abstractions to create state charts. The approach was evaluated on a code snippet from automotive industry. However, the heuristics only include some specific patterns that specifically suit the automotive case study. Therefore, they cannot be used for systems from other domains. Furthermore, the heuristics must be chosen very carefully and by domain experts. Jiresal et al. also mentioned that better heuristics require user interaction, but they did not implement that.

Symbolic execution is used by Kung et al. (Kung et al., 1994) and Sen and Mall (Sen and Mall, 2016). The techniques in these approaches were presented in Section 2, since our interactive approach builds upon this work. Our experiments with these approaches showed that they fail to extract understandable models from typical embedded software: Even small functions can result in hundreds of states and transitions. Our focus in this paper is to improve these approaches towards extracting understandable state machines from real-world software systems.

8 CONCLUSION AND FUTURE WORK

In this paper, we have introduced several options for supporting state machine extraction from code by user interaction. Our case study showed that each of these techniques can help to reduce the complexity of the resulting state machines. It is then up to the user to decide which combination of techniques and parameters is most useful for gaining an understanding of the underlying system.

We are aware that our case study is just a first step towards comprehensive evaluation of our interactive approach. A controlled experiment about the effectiveness of this approach with a larger number of experts is our essential future work. In this experiment, one group of developers will be asked to use our approach and a control group will use other state-of-the-art tools to perform a set of program understanding tasks. Comparing the results from these groups will give a well-grounded evaluation of the approach. In addition, we plan to evaluate each scenario separately to see the benefit of every technique. We want to find out which scenario is most helpful given different situations. We also want to assess the developers' effort for applying the various selection and filtering features. The first results are very promising.

Another issue that has to be addressed in the future is the complexity of transition conditions. We have concentrated on states and transition structure so far. However, transition conditions are also important for understanding. The conditions that result from the extraction process are usually extremely complex and not understandable for humans. Also, reducing the number of states obviously makes state machine models more understandable. However, the information that was previously contained in the state invariants will not get lost, but it will move into the transitions, which may make those more complex. We are currently investigating approaches for reducing this complexity. For example, not all information in these conditions is relevant. This could lead to another interactive extension.

Other opportunities for future research include additional interactions, automated identification of effective constraints, and runtime optimization for the SMT solver. We also want to apply similar interactions to extraction of other kinds of models that can support developers in reengineering tasks.

In summary, adding interaction to state machine mining appears to be a promising approach towards practically usable support for program comprehension and migration towards model-based development.

REFERENCES

Abadi, M. and Feldman, Y. A. (2012). Automatic recovery of statecharts from procedural code. In *Int'l Conf. on Automated Software Engineering*, pages 238–241.

Ammons, G., Bodík, R., and Larus, J. R. (2002). Mining specifications. In *Proc. 29th Symp. on Principles of Programming Languages*, pages 4–16.

Broy, M., Kirstan, S., Krcmar, H., Schätz, B., and Zimmermann, J. (2013). What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, Applications*, pages 310–334.

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 *Proc. of 22nd ICSE*, pages 439–448.

Eisenbarth, T., Koschke, R., and Vogel, G. (2005). Static object trace extraction for programs with pointers. *Journal of Systems and Software*, 77(3):263–284.

Fjeldstad, R. K. and Hamlen, W. T. (1984). Application program maintenance study: Report to our respondents. In *Proc. GUIDE 48*.

Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: directed automated random testing. In *Proc. of PLDI*, pages 213–223.

Hoffmann, A., Quante, J., and Woehrle, M. (2016). Experience report: White box test case generation for automotive embedded software. In *Proc. of 9th Int'l Conf. on Software Testing, Verification and Validation Workshops, TAIC-PART Workshop*, pages 269–274.

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*.

King, J. C. (1976). Symbolic execution and program testing. *Journal of the ACM*, 19(7):385–394.

Kung, D. C., Suchak, N., Gao, J. Z., Hsia, P., Toyoshima, Y., and Chen, C. (1994). On object state testing. In *Proc. of 18th Int'l Computer Software and Applications Conference (COMPSAC)*, pages 222–227.

Miranda, D., Genero, M., and Piattini, M. (2005). Empirical validation of metrics for UML statechart diagrams. In Camp, O., Filipe, J. B. L., Hammoudi, S., and Piattini, M., editors, *Enterprise Information Systems V*, pages 101–108. Springer Netherlands.

Nejmeh, B. A. (1988). NPATH: A measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200.

Prywes, N. and Rehmet, P. (1996). Recovery of software design, state-machines and specifications from source code. In *Proc. of 2nd Int'l Conf. on Engineering of Complex Computer Systems*, pages 279–288.

Quante, J. (2016). A program interpreter for arbitrary abstractions. *16th Int'l Working Conference on Source Code Analysis and Manipulation*, pages 91–96.

Roehm, T., Tiarks, R., Koschke, R., and Maalej, W. (2012). How do professional developers comprehend software? In *Proc. of 34th ICSE*, pages 255–265.

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

Somé, S. S. and Lethbridge, T. (2002). Enhancing program comprehension with recovered state models. In *10th Int'l Workshop on Program Comprehension (IWPC)*, pages 85–93.

Walkinshaw, N., Bogdanov, K., Ali, S., and Holcombe, M. (2008). Automated discovery of state transitions and their functions in source code. *Softw. Test. Verif. Reliab.*, 18(2):99–121.

Walkinshaw, N. and Hall, M. (2016). Inferring computational state machine models from program executions. In *Proc. of Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 122–132.

Wang, S., Dwarakanathan, S., Sokolsky, O., and Lee, I. (2012). High-level model extraction via symbolic execution. Technical report, Univ. Pennsylvania, Dept. of Computer and Information Science. MS-CIS-12-04.

Weiser, M. (1981). Program slicing. In *Proc. of 5th Int'l Conf. on Software Engineering*, pages 439–449.

Xie, T., Martin, E., and Yuan, H. (2006). Automatic extraction of abstract-object-state machines from unit-test executions. In *Proc. 28th ICSE*, pages 835–838.