IEICE TRANS. INF. & SYST., VOL.E103–D, NO.8 AUGUST 2020

PAPER *Special Section on Formal Approaches*

Model Checking of Automotive Control Software: An Industrial Approach

Masahiro MATSUBARA[†,††], ***Nonmember and*** **Tatsuhiro TSUCHIYA**[††a)], ***Member***

SUMMARY In automotive control systems, the potential risks of software defects have been increasing due to growing software complexity driven by advances in electric-electronic control. Some kind of defects such as race conditions can rarely be detected by testing or simulations because these defects manifest themselves only in some rare executions. Model checking, which employs an exhaustive state-space exploration, is effective for detecting such defects. This paper reports our approach to applying model checking techniques to real-world automotive control programs. It is impossible to directly model check such programs because of their large size and high complexity; thus, it is necessary to derive, from the program under verification, a model that is amenable to model checking. Our approach uses the SPIN model checker as well as in-house tools that facilitate this process. One of the key features implemented in these tools is boundary-adjustable program slicing, which allows the user to specify and extract part of the source code that is relevant to the verification problem of interest. The conversion from extracted code into Promela, SPIN's input language, is performed using one of the tools in a semi-automatic manner. This approach has been used for several years in practice and found to be useful even when the code size of the software exceeds 400 KLOC.

key words: *model checking, program slicing, automotive control systems, SPIN model checker*

1. Introduction

In automotive control systems, potential risks to system safety have been increasing because programs are becoming increasingly larger due to advances in electric and electronic control, communication, diagnosis, and other functions. To ensure safety, testing or simulations are commonly used for defect identification. However, it is difficult to detect corner case defects using these ordinary techniques due to the limitation of test coverage achieved by them. In particular, race conditions caused by concurrent execution of hardware devices or of software processes are very difficult to find. In addition to these ordinary techniques, the development of automotive control systems usually involves long-term testing performed under an environment similar to the actual running environment. This is performed by means of HILS (Hardware-In-the-Loop Simulation) or virtual simulation, aiming to find corner cases. However, such long-term testing still provides no guarantee of finding corner case defects. Although static analysis of source code is an effective way to find errors [1], static analysis of concurrent software is still a challenge in terms of precision and scalability [2].

To solve these shortcomings, we adopt an approach that makes full use of model checking [3] in the development cycle of automotive control software. Model checking is a formal verification method which is based on mechanical state exploration. An advantage of model checking over static analysis is a richer set of properties that can be verified. Also, model checking has proved to be very useful in discovering rare and subtle scenarios which otherwise could not be detected in many areas including communication protocols and hardware designs. In the context of the development of automotive control software, model checking could be used for verification of control models or software specifications; but in this paper, we focus on verification of source code.

Applying model checking to automotive control software programs is not an easy task. The most serious problem is state space explosion where an enormous number of states lead to a shortage of memory or time [4]. Hence it is necessary to construct a tractable input model of a model checker from the software. However, a model at an appropriate accuracy level is hard to construct either manually or automatically because source code of automotive control software is very complex.

Our approach addresses this difficulty by allowing the verifier with design knowledge to intervene in adjustment of the accuracy level, as well as to automate the rest part for efficiency. The verifier makes the adjustment of the accuracy level by specifying the part of the source code that is extracted and model checked. In order to perform this process in a flexible and efficient manner, a computer-aided solution is needed that can help the verifier to recognize the part of the source code that is relevant to the property to be verified and to judge how completely the relevant part is transformed into a model. A rule of thumb of this decision is that the specified part of the code is large enough to contain functionality that could not be tested by unit testing but small enough to avoid state explosion. Since such a decision is hard to be automated, manual intervention is essential.

To satisfy these requirements, we have developed a tool that visualizes the structure of the code as a form of a graph or tree and helps specify and extract relevant part of the code using a code slicing technique, which we call boundary-adjustable slicing. We have also developed a tool that converts an extracted code into a model in Promela, the

Manuscript received October 8, 2019.
Manuscript revised February 6, 2020.
Manuscript publicized March 30, 2020.

[†]The author is with Hitachi Automotive Systems, Ltd., Hitachinaka-shi, 312–8503 Japan.
[††]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: t-tutiya@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2019FOP0002

input language of the SPIN model checker [5]. The Promela model obtained in this way can readily be verified using this model checker.

In this paper, we describe our approach with an emphasis on these tools that we have developed. The rest of the paper is structured as follows: Sect. 2 explains the proposed approach and the tools used in the approach. Section 3 shows a program slicing technique that is one of the key technologies in our approach. Section 4 describes how models of automotive control systems are constructed from extracted code. Section 5 presents a typical use case of our approach and summarizes some experiences cases. Section 6 describes related work. Finally, Sect. 7 concludes the paper.

2. Methodology and Tools

2.1 Characteristics of Automotive Control Software

Figure 1 shows a typical structure of an automotive control system. The system connects several Electronic Control Units (ECUs) with a LAN. Software runs on the ECUs while controlling the sensors and actuators connected to them. Automotive control software has many distinctive traits as listed below.

- ECUs are equipped with CPUs (central processing units) and ASICs (application-specific integrated circuits). CPUs are often multi-core. Software runs in parallel on CPUs and ASICs.
- Automotive control software constitutes a concurrent system with hard real-time cyclic tasks and interruptions. Task invocations and interruptions occur by excitation from peripherals (e.g. timer) of a microcontroller.
- Modular software components and legacy complex device drivers are placed on a hierarchical architecture. They are tightly connected aiming at high performance and less resource consumption. As a result, the effects of a failure can easily propagate to other parts of the system and lead to malfunctions.

Because of these characteristics, even carefully designed automotive control software code cannot be perfectly free from defects, especially concurrency defects such as race condition. Therefore, programs should be verified at the source code or binary code level. Simulations, such as HILS, are effective at finding concurrency defects, but only to some extent, because the execution timing of tasks, interruptions, or hardware has an enormous number of combinations and simulations can exercise only a small fraction of them. Furthermore, single automotive control software runs on many cars for a long time. This means that a defect that has negligible occurrence probability in isolation might cause a significant safety problem. Such defects are often triggered by particular input combinations or execution timing of concurrent processes. Model checking is a viable solution to solve these problems, as it can verify all possible behaviors of the software under test. We target model checking of source code rather than binary code, because concurrency defects can be found in source code. The verification method must be able to handle the following characteristics of source code of automotive control software.

- Source code is usually written in the C language, because of its high processing speed and capability of controlling low-level I/O operations.
- Global variables are shared among tasks and interruptions. Race conditions between them must be avoided carefully by using buffers, mutex, or task prioritization.
- Programs are large in size and have high complexity. One controller in a powertrain system or chassis control system often consists of more than 100k lines of code.

However, the verification method is not required to handle all characteristics of the C language. For example, dynamic process creation and dynamic memory allocation are out of the scope because they are seldom used in automotive control software. Pointer variables need to be considered but pointed addresses can be statically analyzed since no dynamic memory allocation is used.

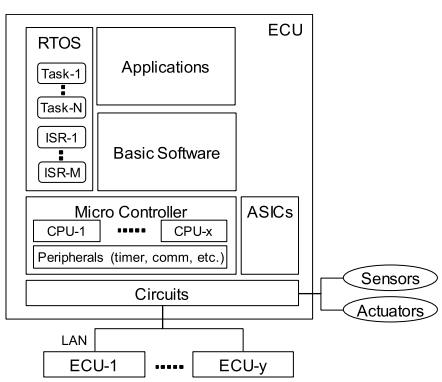

Fig. 1 Typical structure of automotive control system

2.2 Model Construction Procedure with the Tool Chain

Generally, the process of applying model checking to a real-world problem consists of two phases: constructing a model of (part of) the system to be verified and executing a model checker against the model obtained in the first phase. To deal with automotive control software which has the above characteristic, we base our methodology on SPIN, one of the best known model checkers. The input model of SPIN is specified in the Promela language. We think that the choice of SPIN is natural and appropriate, because Promela has C-like syntax and provides built-in support for modeling of concurrent processes. In addition, SPIN has proved to be very powerful in model checking nontrivial models that arise from real-world problems. Recent examples include [6]–[8].

In spite of the high verification performance of SPIN, it is far from practical to model check the entire automo-

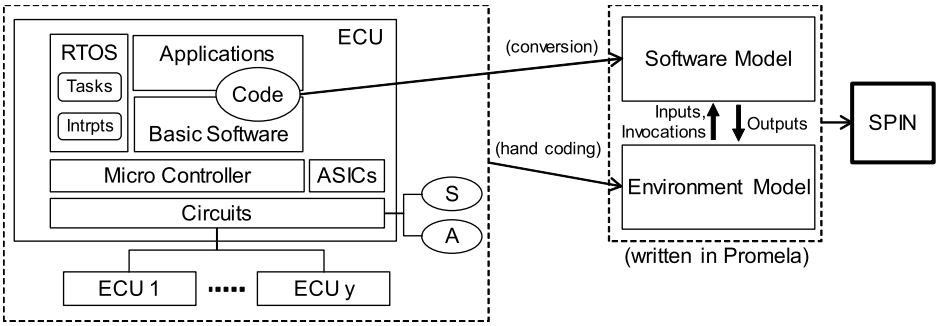

Fig. 2 Model structure. The model consists of a software model that represents an extracted code and an environment model that simulates the input to the software part

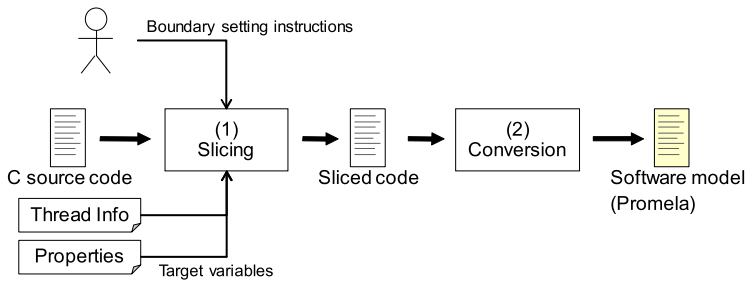

Fig. 3 Tool functions for software model generation

tive control program at once. It is thus required to extract the part of the source code that is relevant to the problem in hand and then describe the extracted part in Promela. It is also necessary to model the environment that interacts the system or the extracted part of the code. Hence, conceptually, we treat a Promela model that represents the system under verification as a combination of a software model and an environment model as shown in Fig. 2.

The software model represents the software behavior based on the extracted code. The environment model simulates the input of software model including invocation of tasks or interruptions, or reaction to the output of the software model.

It would be tedious and error-prone to manually construct the software model by examining the program code. To facilitate this process, we developed using the Java language a tool for computer-aided model construction. This tool has two core functions. The first one is program slicing, which is a function that extracts part of source code that is relevant to specified variables that are used in a property or an assertion. The second function is model conversion which translates the extracted C source code into Promela. Figure 3 outlines the process for software model generation in our methodology.

In the course of our attempt to apply model checking to real-world problems, we found that simply using code slicing was not very useful, because the extracted part could be too large to be converted into a model amenable to model checking. To address this problem, we developed a program slicer that implements what we call *boundary-adjustable program slicing*. In boundary-adjustable program slicing, the user can set and adjust the boundary on the program to which slicing is performed. The boundary is adjustable in an interactive and iterative manner. How boundary-adjustable program slicing is implemented in our tool will be described in Sect. 3.

Our tool also supports semi-automatic conversion from the extracted code to Promela. The conversion process uses Abstract Syntax Tree (AST) as intermediate representation between the C language and Promela. The code is parsed into AST and then converted to Promela. The model construction process, including code conversion and integration with the environment model, will be described in Sect. 4.

3. Boundary-Adjustable Program Slicing

3.1 Program Slicing Overview

Program slicing [9] is the technique to extract parts of a program that (potentially) affect the values computed at some point of interest. Slicing is used for many purposes, including debugging, regression testing, and software maintenance [10], [11]. In our context, slicing is used to extract a part of program code that is relevant to the verification property of interest, aiming to obtain a model of reasonable size.

Slicing is very useful to reduce the size of the code to be verified. However, the size of extracted code is still often too large to be model checked, especially when the input program is large, like automotive control systems. Some study reported that the average size of sliced code was 30% of the input program on average [12]. We take an approach,

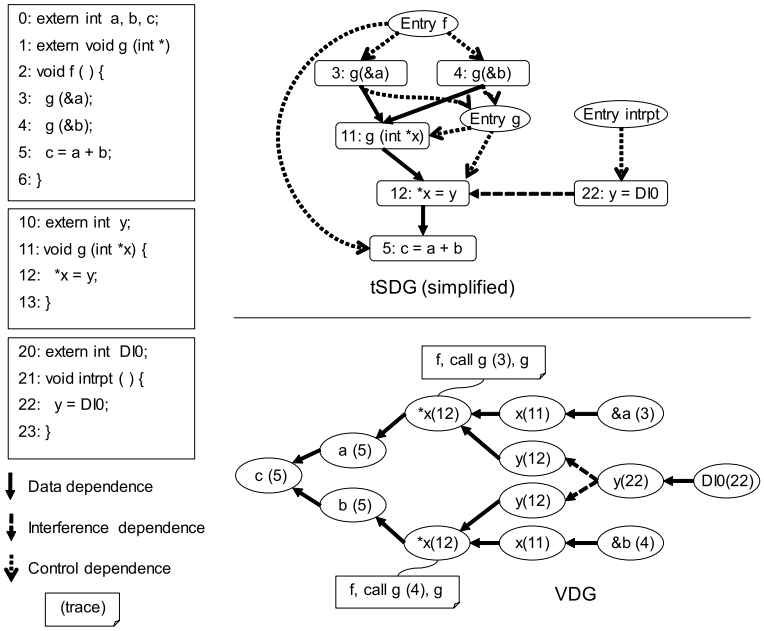

Fig. 4 tSDG (above) vs. Variable Dependence Graph (bottom)

we named boundary-adjustable program slicing, to address this problem. This approach allows the user to set and adjust boundaries to which program slicing is executed.

Program slicing often operates on a graph representation of the target program to track control and data flows. In our method, *Variable Dependence Graphs (VDGs)* are used for this purpose.

3.2 Variable Dependence Graph (VDG)

In order to analyze concurrent programs, it is required to track interference dependence [13], in addition to control and data dependence. Interference dependence is data dependence that occurs between different threads. Graph representations that can model these dependencies include threaded Program Dependence Graphs (tPDGs) [13], threaded System Dependence Graphs (tSDGs) [14], and threaded Interprocedural Program Dependence Graphs (tIPDGs) [15].

These graph representations are useful for automatically analyzing programs but take almost no considerations about the easiness of manually operating on the graph. Hence we developed a new graph representation which we call *Variable Dependence Graph (VDG)*.

VDGs are different from these existing graph representations which are basically extensions of well-known Program Dependence Graphs (PDGs) [16], [17]. A VDG is a directed graph $G = (V, E)$ such that: a node n in V is either of variable, constant, or control statement (e.g. "if", "while"), whereas data dependence exists from s (either variable or constant) to t (variable), or control dependency exists from s (control statement) to t (variable or constant) if an edge (s, t) exists in E. When a node s represents a variable or a constant in a condition expression of a control statement, control dependency also exists from s (variable or constant) to t (control statement). Other control dependence that exists in other graphs like tSDG is omitted. Another type of interdependencies, called interference dependencies, are also indicated by edges.

Each node in a VDG is associated with its position (file, line, and column) as well as its calling context. Calling context enables precise dependence analysis, thus reducing unnecessary code [18]. Therefore, a variable occurring in the same position but with different calling contexts is expressed as different nodes. In contrast to a VDG, a statement with different calling contexts is represented by one node in a tPDG or tSDG; thus one has to consult another complementary graph to distinguish different calling contexts. Figure 4 shows examples of tSDG and VDG for the same program. In the VDG, `x` in statement 12 is duplicated to two nodes, one corresponding to the call to `g` from statement 3 and the other corresponding to the call to `g` from statement 4.

Interference dependencies of global variables shared between different tasks or interruptions are found between two nodes in a VDG such that a variable is referred at one node and is assigned a new value at another node in a different thread. In the VDG of Fig. 4, this kind of dependencies are depicted by dashed lines; the two nodes for variable `y` at line 12, each corresponding to one of the two calls to function g, have interference dependencies from line 22 in the interruption (lines 20-23).

In VDGs a node for a pointer variable is associated with the pointed variable that is equivalent to the address resulted from a point-to analysis. Interference dependence from a pointer variable to the pointed variable is also expressed in VDGs.

In our tool, VDGs are constructed by analyzing programs as follows. Dependence within each task is analyzed

first, and then interference dependence is analyzed using the approach proposed by Krinke [15]. Krinke's algorithm is suited for slicing concurrent programs that use shared global variables and do not create new process in run time. Point-to analysis is carried out for each thread, by following the data dependence in a thread. To simplify the analysis task, the results of point-to analysis are not shared among threads. From our experiences this suffices because we have never encountered programs where different threads set the value (address) of a pointer. When priorities are given to the threads of the programs, interference dependencies are generated comparing the priorities of the threads. Dependencies within data members of C unions are also retrieved. Dependencies in a cyclic task are analyzed as if the code constituted a body of a big loop. The analysis is skipped for a control flow when a recursion is detected.

To achieve high performance, we adopt parallelization using multiple cores of a CPU in two places. First, parsing is performed in parallel for each file. Second, dependence analysis is carried out in parallel for each thread.

The parallelization provides some speedup; but in order to handle large programs in automotive domain, we needed more powerful techniques. To this end, we developed a VDG construction program by extensively using hashes, instead of standard graph manipulations. This approach can reduce a large amount of time in practice, because using a hash, operations on graphs that would take quadratic time or more in theory can often be performed in a constant time. However, this comes with the cost that backward slices cannot be extracted during constructing a VDG (or other program graph representations). Also, extracting full backward slices (which are *Variable Dependence Trees (VDTs)* (Sect. 3.3) in our case) from the VDG obtained may require a very large amount of time when the slices are large and especially have a large number of branches. Boundary-adjustable program slicing solves this problem by allowing the user to set boundaries near the variable used as the slicing criterion. This can delay the process of extracting large VDTs until it is actually needed. In fact, as stated in Sect. 2.2, large slices are not very useful for model checking purposes; therefore, this time-consuming process usually never takes place.

Using these techniques as well as many other optimizations, the tool is currently capable of dealing with programs over 400kLLOC (Logical Lines of Code) on a common PC. To our knowledge, there are no other program slicing tools that can handle programs of this size.

3.3 Variable Dependence Tree (VDT) and Boundary-adjustable Program Slicing

Boundary-adjustable program slicing operates on a *Variable Dependence Tree (VDT)*, an alternative representation of (part of) a VDG $G = (V, E)$. A VDT is a rooted tree $T = (V', E')$, where V' is the set of nodes and E' is the set of directed edges. Each node in V' corresponds to a node in E, while each edge in E' corresponds to an edge in V. The root

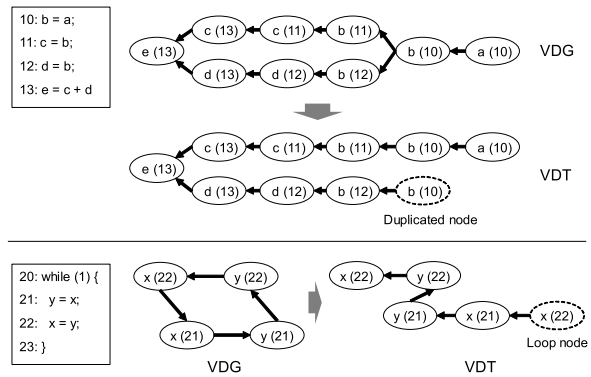

Fig. 5 Conversion from VDG to VDT

node represents the variable that serves as the slicing criterion. Typically, the variable is the one involved in a property or assertion that needs to be verified. There are two types of VDTs: goal VDT and start VDT. In a goal VDT, the root node has incoming edges from its child node, and the child nodes have incoming edges from their child nodes, and so on. In a start VDT, on the other hand, each node has outgoing edges to its child node. The program slice obtained from a start VDT is a forward slice, which contains those that are affected by the criterion. A goal VDT, on the other hand, yields a backward slice, which contains the statements that can have some effect on the slicing criterion. In the following of this section, we limit our description to goal VDTs for presentation simplicity.

To convert a VDG into a VDT, two types of nodes are introduced. One is duplicated nodes and the other is loop nodes. A duplicated node appears as a leaf of a VDT to show that a node with the same position and same calling context has already appeared elsewhere in the tree. A loop node also appears as a leaf of the tree to indicate that a node with the same position and same calling context has already appeared as its ancestor. Examples of goal VDTs with a duplicated node or loop node are shown in Fig. 5. Node b at line 10 in the above VDG of this figure is separated into two nodes in the VDT, one of which is a duplicated node. Node `x` at line 22 in the bottom VDG is divided into two nodes in the VDT on the right. One of the two nodes is a loop node, indicating a cyclic dependence. Figure 6 shows a screen shot of the tool (above) and how a VDT is displayed in the tool (bottom).

Our tool performs dependence analysis in two stages, that is, during the construction of a VDG and during the extraction of VDTs. A VDG is constructed only once for a given program, whereas VDTs are extracted every time when a variable is selected as the slicing criterion (i.e. the root node) by the user. Once a variable is selected, the VDG is traversed to find the nodes that match the variable. The nodes thus found serve as the root nodes of VDTs. VDTs are constructed by extracting the descendant nodes of these root nodes.

The purpose of using VDTs is to implement boundary adjustable slicing; that is, to allow the user to interactively

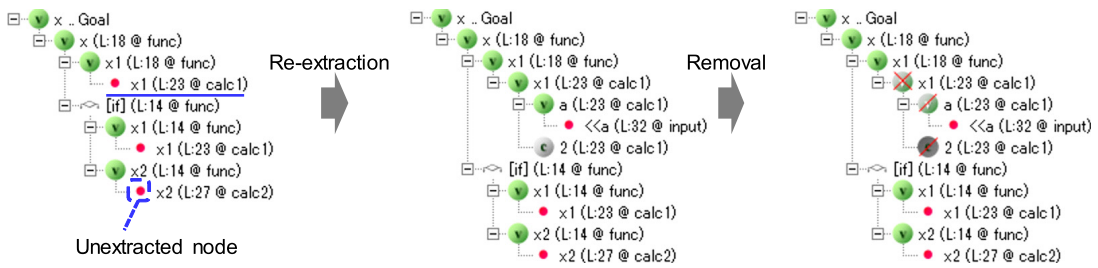

Fig. 7 Limited extraction and re-extraction of VDT

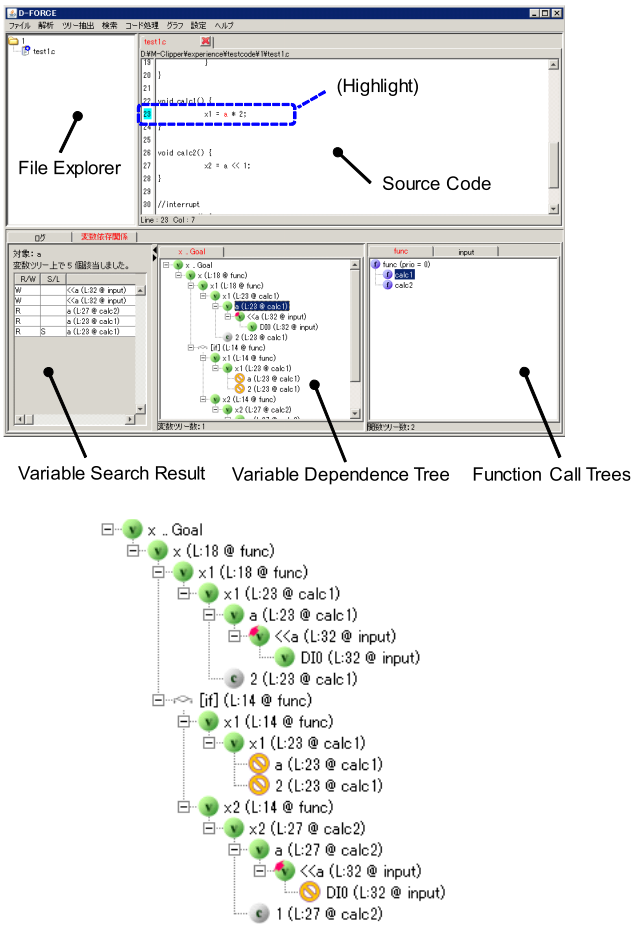

Fig. 6 Screen shot of the tool. The whole window of the tool (above) and a VDT (bottom)

adjust the part of code to be verified. Since the VDTs represent the designated part of the programs, the construction of VDTs from a VDG should accept user's control. To this end the tool constructs each VDT in two steps. In the first step, descendant nodes are extracted by automatically traversing the edges in the VDG. (It should be noted that descendant nodes in a goal VDT are ancestor nodes in the corresponding VDG.) The traversal tracks back if it visits a node belonging to a function different from the slicing criterion variable. Once the descendant nodes are extracted within the C function where the slicing criterion variable is located, the control is handed over the user. In the second step, the user repeats extending the VDT by adding a descendant node until the VDT contains all nodes of interest. The user is also allowed to remove VDT nodes to shrink the VDT and program slices obtained from it. The user can perform both addition and removal of nodes interactively through the tool's interface, as shown in Fig. 7. Boundary adjustment is achieved with such gradual addition and removal of nodes.

A program slice is constructed based on a VDT. A statement remains in a program slice if it has at least one corresponding node in the VDT. Control structures that surround the remaining statements, e.g. functions or other control constructs, are also preserved in the slice. As stated above, a goal VDT and start VDT produce a backward slice and forward slice, respectively. In most of the cases we have experiences, backward slicing was used for model checking.

A root node of a VDT is selected based on the property to be verified. When only a single variable (say v) occurs in the LTL formula or assertion, the node that corresponds to v becomes the root of a goal tree. If multiple variables are involved, a node corresponding to one of these variables is selected as the root so that the remaining variables can occur somewhere in the tree paths. Boundaries need to be set so that the variables holding output values can be included in the extracted code. For start trees, a variable representing output is selected as the root and boundaries are set outside the variables involved in the property to be verified.

The reason we use VDGs and VDTs, instead of other graph representations, such as tSDG, is that we think VDG/VDT are suitable for software designers or verifiers to track and comprehend the dependencies in the program from the variables' point of view. The visualization provided by VDGs/VDTs focuses the dependencies between variables by removing other information, such as operators that are contained in the statements corresponding to their nodes. However, the user might need to see the whole statements. To deal with such cases, the tool provides a feature that allows the user to designate a node and view the full statement corresponding to the node.

4. Constructing Promela Models

4.1 SPIN Model Checker and Promela Language

Model checking automatically determines whether given properties are satisfied by the system under verification. The system is usually modeled as a finite state machine. Model

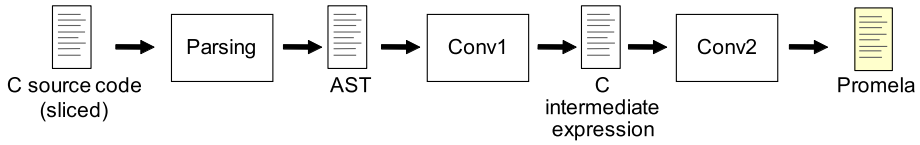

Fig. 8 Conversion steps from C to Promela

checkers perform verification by exhaustively searching the state space of the state machine model. In our approach we use SPIN, one of the best known model checkers. SPIN takes as input a program written in Promela (Process Meta Language). In Promela, the behavior of the given system is represented as one or more concurrent processes. The semantics of Promela assumes that the Promela processes run concurrently in an interleaving manner; that is, exactly one single atomic step of a Promela process is executed at a time.

A Promela process is composed of data objects, nondeterministic constructs, and communication primitives. Processes can communicate via synchronous and asynchronous message passing with buffered channels or shared memory. Properties to be verified are specified by assertions, Linear Temporal Logic (LTL) formulas, or special finite state automata called never claims. LTL formulas and never claims can express general safety and liveness properties. SPIN performs on-the-fly verification and employs many useful state search and compression techniques.

4.2 Software Model

Once a program slice has been extracted using the method described in Sect. 3, a software model is then constructed by converting the C code extracted by program slicing into Promela. This converting process results in a single or multiple Promela processes. A typical case where multiple Promela processes are obtained is when the extracted C code contains calls to functions that are invoked as individual tasks. However, Promela lacks some basic constructs or variable types in C; thus some techniques are needed as follows.

- C functions are converted to inline macros of Promela.
- All C local variables are converted into global variables in Promela, Promela supports local variables, though. The reason for this is that global Promela variables provide better expressiveness and flexibility, because the scope of a local variable in Promela is the complete body of a process declaration. Global Promela variables representing C local variables in functions are replicated and renamed for each call.
- Accesses to the locations specified by pointers of the C program are converted into operations on global Promela variables.
- Nested assignments or function calls are divided into multiple statements.

These conversions are performed by following the control flow semantics of C, thus preserving the behavior of the extracted code in terms of control flow. We do not consider variables of float or double type, since most of the C programs in our traditional product domain contain no variables of these types. This is because these C programs use integer variables to manipulate real numbers to avoid unexpected floating-point errors.

The conversion process runs in three steps, as shown in Fig. 8. A given C program is first parsed into an Abstract Syntax Tree (AST). Then the AST is translated into an intermediate expression of the C language and then into Promela. The intermediate C expression is a simplified AST, where elements that are not relevant to the C-Promela conversion are removed and some C statements are simplified so that they can be directly converted in Promela. For example, C statement `x = (a = b + c) * d;` is represented as `a = b + c; x = a * b;` in the intermediate expression.

These three steps of conversion are almost fully automated, except the third step where manual intervention may be needed in some cases. A typical example of such a case is the conversion of operations on a pointer variable into those on global Promela variables. During traversing the C intermediate expression, the tool issues warnings when it encounters elements that require manual handlings.

Even after a Promela program has been obtained, further manual modifications can be useful in some cases. For example, if the state space of the model turns out to be too large to model check, simple optimizations, such as constant propagation or similar techniques, may help solve the problem to some extent.

4.3 Environment Model

The outside environment that interact with the software model needs to be abstracted and modeled as either 1) individual Promela processes or 2) Promela code fragments that are inserted into Promela processes of the software model, or both. We refer to this overall Promela code that models the outside environment as the environment model. Unlike the software model, the environment model is constructed manually.

Figure 9 illustrates the boundary between the software model and the environment model. A boundary on the VDT becomes the boundary between the two models. In the figure, a boundary is placed between line 3 and line 4 in the C program. As a result, variable `z` at line 3 is not included in the extracted code and thus should be treated in the environment model.

One of the main roles of the environment model is to feed input values to variables of the software model, such as variable `z` in Fig. 9. The ranges of input values fed by the

```
void control()
{
  if (!err) {
    cmd = strk * C_STRK2CMD;
  }
  else ...
}
```

(a) Extracted C code

```
int cmd, strk; bool err;
inline control()
{
  if
  :: (!err) -> cmd = strk * C_STRK2CMD
  :: else -> ...
  fi
}
```

(b) Software model in Promela

```
inline pedal()
{
  if
  :: ndval(strk, 0, 5, 10); err = false
  :: strk = 0; err = true
  fi
}
```

(c) Environment model in Promela

```
int cmd, strk; bool err;
inline control()
{
  if
  :: (!err) -> cmd = strk * C_STRK2CMD
  :: else -> ...
  fi
}

inline pedal()
{
  if
    :: ndval(strk, 0, 5, 10); err = false
    :: strk = 0; err = true
  fi
}

active proctype main() {
  pedal();
  control()
}
```

(d) Resulting Promela program

Fig. 10 Extracted C code, software model, environment model, and resulting Promela program

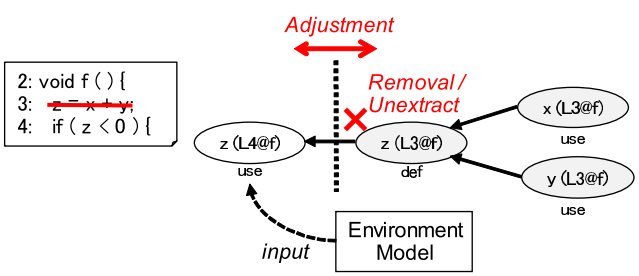

Fig. 9 Relation of environment model and boundary on VDT

environment model need to subsume the input value ranges of the program under verification because otherwise possible behaviors of the program could be missed in the final model. By providing such input values by the environment model, we can over-approximate the behaviors of the program. The nondeterministic construct of Promela can be nicely used for this purpose. The following Promela code is an example of a macro that can be used to select one of the actual parameters non-deterministically and assign it to the variable designated by the first parameter.

```
#define ndval(var, vl0, vl1, vl2) \
if :: var = vl0 :: var = vl1 :: var = vl2 fi
```

Now suppose that the C code shown in Fig. 10 (a) has been obtained using boundary-adjustable program slicing. The Promela model obtained from this code looks like as shown in Fig. 10 (b). Variable `strk` takes brake pedal stroke amount as input. Also variable `err` is used as a flag that signifies the occurrence of an error.

Its environment model shown in Fig. 10 (c) provides values chosen non-deterministically to these variables. The whole Promela process is obtained by combining these models, as shown in Fig. 10 (d).

A technique that we found to be useful is to provide input values that are outside the specification range; this allows us to observe the robustness of the software to unexpected input or hardware malfunctions.

Sometimes, hardware components, such as peripherals of a micro controller or ASICs, need to be represented in the environment model. A typical case is when one needs to verify the correctness of the internal state of a hardware component that is controlled by the software. Such a hardware component can be modeled as, for example, an individual Promela process where input and output between the hardware and software are represented in the form of reads and writes of Promela global variables.

Finally, we create a single Promela program by combining the software and environment models. In doing so, we need to model the scheduling policy of tasks, because, as stated in Sect. 2, a program is executed on a Real-Time

Operation System (RTOS) in the form of a set of cyclic tasks and/or interruptions. In practice, the well-known rate-monotonic scheduling algorithm is usually used. SPIN of version 6.2 or later implements the task scheduling algorithm [19]. This feature allows the user to assign scheduling priorities to Promela processes and to invoke the processes periodically as if they were periodic real-time tasks. When the extracted code runs in parallel on different cores of a CPU, the priorities of Promela processes are set to the same value in order to represent the physical concurrency.

5. Practical Considerations

5.1 Typical Scenario

Here we describe a typical scenario of using the presented approach based on our real-world experience: specifically, the experience of verifying diagnosis software for a power supply IC. The software remotely monitors the power supply IC through a serial communication link. In the testing phase of the whole system, we encountered a possible error of the IC; that is, the software signaled an error. However, no fault was found in the post-analysis and no error could be reproduced either.

In such a case, model checking can be useful to investigate the possible fault. Using our approach, a Promela model is built as follows.

1. Select the variable that represents the diagnosis result as the root of a VDT.
2. Perform dependence analysis to obtain a program slice in the form of a VDT.
3. Set the boundary on the VDT such that the code part used for communication between the IC is excluded. This greatly reduces the code size extracted and, in turn, the resulting software model, as the complex processing of the serial communication driver is discarded.
4. Construct manually an environment model to express the behavior of the IC.

In our real-world case, model checking revealed a subtle situation where a mismatch of status recognition could occur. Specifically, it was found that the error can happen if the following two events happen almost simultaneously: 1) the IC changes its state and 2) the diagnosis software enters the sleep mode and returns to the normal mode immediately.

This result was obtained by verifying the program against the property that the error flag never turns on as a result of diagnosis, which is a property that is supposed to hold in error-free runs. The property can be represented as LTL formula `[](!p)` where `p` expresses that the value of the error flag variable is true. Table 1 shows the sizes of the source code, software model, and environment model.

Table 1 Sizes of source code and Promela models

Total size of source code	Approx. 130 kLLOC
Size of software model	600 LOC
Size of environment model	150 LOC

5.2 Division of a VDT

In practice, a cause of malfunction is not always found within the initial VDT. In that case, a VDT should be enlarged to include more code into the software model. However, this may lead to state explosion. A divide-and-conquer strategy can be helpful to solve this situation. That is, a VDT is divided into several subtrees such that a variable that is designated as a boundary of one subtree is selected as the root of another subtree. We call the subtrees of a VDT sub-VDTs.

A major problem arising when using this strategy is how to divide the VDT into sub-trees. Basically, the dependency of different subtrees should be minimized so as to allow the verifiers to verify each subtree in as much isolation as possible. Also doing so requires feedbacks from the software designers. To support the dividing process, we use a notation which we call a function dependence tree (FDT) (Fig. 11). Each of the non-root nodes in an FDT corresponds to a sub-tree of the VDT. The root node of an FDT represents the variable under verification, which corresponds to the root node of the VDT. A non-root node is labeled with the name of its representative function and the name of the task or the interrupt service routine that the function belongs to, whereas an edge is labeled with a variable that is used to interact the codes represented by the two end nodes of the edge. FDTs help both software designers and verifiers to understand the planned division of a VDT and facilitate discussion among them.

5.3 Experienced Cases

The model checking approach has been successfully applied to many automotive control software products since 2012, after a few years of basic studies. Table 2 summarizes some experienced cases. In this table, the row named Existing defect indicates whether or not the existence of a defect was known before verification. Completion of verification means that model checking was successful in finding an error or was able to fully explore the state space of the obtained model. The row named Number of divisions shows the number of sub-VDTs obtained by dividing the extracted code to form an FDT (see Sect. 5.2). The functionalities provided by our automotive software products are basically categorized into four types: initialization/finalization (of a

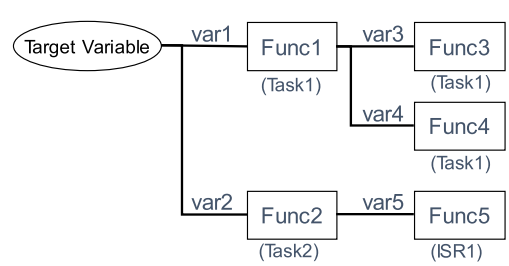

Fig. 11 Function Dependence Tree (FDT)

Table 2 Size of source codes and models

	case 1	case 2	case 3	case 4	case 5
Total size of source code [kLLOC]	129.8	256.3	287.2	411.9	411.9
Extracted code size [kLLOC]	0.92	3.10	24.0	6.79	12.94
Existing defect	Yes	No	No	Yes	Yes
Completion	Yes	No (time limitations)	No (out of memory)	Yes	Yes
Number of divisions	1	1	1	11	7
Detection of errors	Yes	No	No	Yes	Yes
Functionality	Diagnosis	Control	Control	Finalization	Diagnosis

Table 3 Statics on verification of large models

	states	transitions	time [s]	memory
Part of Case 4	1.04×10^7	5.72×10^7	544	6.8 [GB]
Part of Case 5	1.34×10^9	2.86×10^9	38300*	116 [MB]*

* MA (Minimized Automaton) option enabled

task), communication, diagnosis, and control (of actuators).

From the table, it can be seen that the verification was completed for source codes whose size exceeded 100 kLLOC. Cases 4 and 5 dealt with the largest code; but for these cases the whole state space could be explored, since the extracted code was successfully divided into several sub-VDTs. This was possible because the functional logics for the diagnosis and finalization functions were well-separated from each other. For example, in the finalization function of case 4, system shut down occurs due to various factors and different factors are dealt with by different logics.

For Cases 1, 4, and 5, the existence of a defect had already been known. In all these cases, model checking was successfully completed, partly because SPIN stops immediately when it detects a fault. Interestingly, the detected fault for Case 1 turned out not to be an actual one: Further examination revealed that the real fault existed in another part of the software that had not been extracted. In contrast, verification was not completed for Cases 2 and 3. It was impossible to conclude from this result that the extracted codes were defect-free; but the fact that no errors were found until exhausting memory or time provided some confidence in the absence of them. Table 3 shows the statics on model checking of some of the largest models arising in the five cases.

In our several year's experience, most of the root causes of errors we found stemmed from either concurrency (type I) or complex logics of programs (type II). The example described in Sect. 5.1 belongs to Type I. The defects found in Cases 4 and 5 were also of this type. In our experience, hardware models that interact with the input or output of the software model were almost always necessary to detect faults of Type I. A lesson learned from modeling hardware is that this process, in effect, works as code review because constructing a hardware model enforces verifiers to better understand the specifications of the hardware and to make sure that the way the software uses the hardware conforms to the specifications. Conversely, Type II faults are not directly related to concurrency and thus can be detected using a software model consisting of a single process. A usual belief is that the main role of traditional model checkers (e.g., SPIN) is to verify concurrent systems and not sequential programs. From our experience, however, our approach based on SPIN usually works well for detecting non-concurrency faults.

Most of the LTL formulas used to represent properties were in the following forms: `[]p`, `[](!p)` (as in Sect. 5.1), and `[](p -> <>q)`. The former two forms represent invariants. We used these forms to describe properties that should hold between control commands and system output, for example. Assertions can also be used instead of these LTL formulas if the program statements at which safety conditions are supposed to hold are clear. The last form, stating that if `p` holds, then `q` will eventually hold, was used to represent the property that an event leads to a specific state, for example, a wake-up event leads to a state change of the system.

6. Related Work

Program slicing has been used to obtain from program codes to models that are tractable by model checking. It has been reported that model reduction with program slicing has a factor of four improvements for non-trivial model checking and that it is orthogonal to other reduction measures [20]. For example, a tool called Bandera [21] uses program slicing to model check Java programs. This slicing mechanism is used in JPF2 [22], which is also a model checker for Java. To our knowledge, there is no other tool that integrates model checking and program slicing that allows the user to make a boundary-adjustment to the code to be extracted.

The problem of extracting Promela models from C programs is addressed by some studies. Modex/Feaver [23] is a verification tool for C programs. The tool slices source code based on control flow and converts the extracted code to Promela. POM/MC [24] is another tool that can be used to extract Promela models from C codes. This tool performs the conversion from C to Promela in multiple steps where the program under verification is expressed in different intermediate representations. The authors of [24] claim that the multi-step approach provides better flexibility and extensibility to the conversion process. Our tool also uses multiple steps to construct models from programs and thus inherits this desirable property.

Program slicing has a large body of studies. Our slicing tool adopts or extends some of the previous techniques to implement necessary features. There are basically two algorithms for precise slicing for concurrent programs [25], [26], which are that of Krinke [15] and that of Nanda [27]. Our tool employs the former one. The boundary-adjustment feature implemented by our tool can be regarded as generalization of some previous techniques for controlling the ex-

tracted part of programs [28]–[30]. Distance-limited slicing concisely visualizes dependency by allowing the user to designate a distance from a state of interest in a dependence graph to limit graph elements to be extracted [28]. In our tool, the boundary effectively defines this distance. Barrier slicing obtains small slices by prohibiting slice computation from traversing nodes or edges in the dependence graph that are declared to be a "barrier" [29]. In our tool, the boundary works just as a set of barriers. In [30], a technique that allows the user to gradually expand the extracted dependency graph is proposed for the purposes of feature location and partial comprehension of a system. The boundary-adjustment slicing naturally implements this feature.

To obtain performance sufficient for industry use, we have made a large number of small optimizations to the tool's program. Currently there seems to be no other program slicing tools that can handle a concurrent program whose size is over 200k LOC.

Our approach combines program slicing and SPIN, a model checker that uses well-understood and time-proven algorithms. CEGAR is a different approach to model check large-sized source code [31]. CEGAR refers to a class of techniques that automatically iterate refinement of a model from abstract to concrete level, when a counterexample is detected and it is a false positive. BLAST [32] and SLAM [33] are examples of the CEGAR-based tools for model checking of programs. Unlike these modern tools, our approach requires some manual intervention. However, our experiences show that industrial problems can be successfully handled using traditional well-established techniques, namely, program slicing and classical model checking.

7. Conclusion

In automotive control systems, the potential risks of software defects have been increasing because of increasing software complexity. To detect defects that are difficult to find with usual tests or simulations, a practical model checking approach was described in the paper. In this approach, software models are generated from source code using the boundary-adjustable program slicing technique. Modeling techniques for hardware components that interact with software were also presented. We have been applying the proposed approach to several real-world problems with automotive control software for years and found that the approach is useful for industrial use.

References

[1] D. Engler and M. Musuvathi, "Static Analysis versus Software Model Checking for Bug Finding," Verification, Model Checking, and Abstract Interpretation, ed. B. Steffen and G. Levi, Berlin, Heidelberg, vol.2937, pp.191–210, Springer Berlin Heidelberg, 2004.

[2] A. Miné, "Static Analysis of Embedded Real-Time Concurrent Software with Dynamic Priorities," Electronic Notes in Theoretical Computer Science, vol.331, pp.3–39, March 2017.

[3] E.M. Clarke, O. Grumberg, D. Kroening, D.A. Peled, and H. Veith, Model checking, MIT Press, 2018.

[4] M. Mansouri-Samani, P.C. Mehlitz, C.S. Pasareanu, J.J. Penix, G.P. Brat, L.Z. Markosian, O. O'Malley, T.T. Pressburger, and W.C. Visser, "Program Model Checking: A Practitioner's Guide," 2008.

[5] G.J. Holzmann, The SPIN Model Checker - primer and reference manual, Addison-Wesley, 2004.

[6] Z. Chen, Y. Gu, Z. Huang, J. Zheng, C. Liu, and Z. Liu, "Model checking aircraft controller software: A case study," Software - Practice and Experience, vol.45, no.7, pp.989–1017, 2015.

[7] T. Ovatman, A. Aral, D. Polat, and A.O. Ünver, "An overview of model checking practices on verification of PLC software," Software and Systems Modeling, vol.15, no.4, pp.937–960, 2016.

[8] S.T. Hamman, K.M. Hopkinson, and J.E. Fadul, "A Model Checking Approach to Testing the Reliability of Smart Grid Protection Systems," IEEE Transactions on Power Delivery, vol.32, no.6, pp.2408–2415, 2017.

[9] M. Weiser, "Program Slicing," Proceedings of the International Conference on Software Engineering (ICSE '81), pp.439–449, 1981.

[10] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," ACM SIGSOFT Software Engineering Notes, vol.30, no.2, pp.1–36, 2005.

[11] J. Silva, "A Vocabulary of Program Slicing-based Techniques," ACM Comput. Surv., vol.44, no.3, pp.12:1–12:41, 2012.

[12] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," ACM Transactions on Software Engineering and Methodology, vol.16, no.2, pp.1–32, 2007.

[13] J. Krinke, "Static slicing of threaded programs," 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp.35–42, 1998.

[14] D. Hisley, M.J. Bridges, and L.L. Pollock, "Static Interprocedural Slicing of Shared Memory Parallel Programs," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 2, PDPTA '02, pp.658–664, CSREA Press, 2002.

[15] J. Krinke, "Context-sensitive slicing of concurrent programs," Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11), pp.178–187, 2003.

[16] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization," International Symposium on Programming, LNCS, vol.167, pp.125–132, 1984.

[17] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," ACM Trans. Program. Lang. Syst., vol.9, no.3, pp.319–349, 1987.

[18] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," Proceedings of the 14th International Conference on Software Engineering, ICSE'92, New York, NY, USA, pp.392–411, Association for Computing Machinery, 1992.

[19] M. Florian, E. Gamble, and G. Holzmann, "Logic Model Checking of Time-Periodic Real-Time Systems," Infotech@Aerospace 2012, 2012.

[20] M.B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine, "Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs," Tools and Algorithms for the Construction and Analysis of Systems, ed. H. Hermanns and J. Palsberg, Berlin, Heidelberg, vol.3920, pp.73–89, Springer Berlin Heidelberg, 2006.

[21] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code," Proceedings of the 22Nd International Conference on Software Engineering, ICSE '00, New York, NY, USA, pp.439–448, ACM, 2000.

[22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," Automated Software Engineering, vol.10, no.2, pp.203–232, 2003.

[23] G.J. Holzmann and T.C. Ruys, "Effective Bug Hunting with Spin

and Modex," Model Checking Software, ed. P. Godefroid, Berlin, Heidelberg, p.24, Springer Berlin Heidelberg, 2005.
[24] M. Ichii, T. Myojin, Y. Nakagawa, M. Chikahisa, and H. Ogawa, "A Rule-based Automated Approach for Extracting Models from Source Code," 2012 19th Working Conference on Reverse Engineering, pp.308–317, 2012.
[25] D. Giffhorn and C. Hammer, "An Evaluation of Slicing Algorithms for Concurrent Programs," Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), pp.17–26, 2007.
[26] D. Giffhorn and C. Hammer, "Precise slicing of concurrent programs," Automated Software Engineering, vol.16, no.2, pp.197–234, 2009.
[27] M.G. Nanda and S. Ramesh, "Interprocedural Slicing of Multithreaded Programs with Applications to Java," ACM Trans. Program. Lang. Syst., vol.28, no.6, pp.1088–1144, 2006.
[28] J. Krinke, "Visualization of program dependence and slices," IEEE International Conference on Software Maintenance, ICSM, pp.168–177, 2004.
[29] J. Krinke, "Slicing, chopping, and path conditions with barriers," Software Quality Journal, vol.12, no.4, pp.339–360, 2004.
[30] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," Proceedings IWPC 2000. 8th International Workshop on Program Comprehension, pp.241–247, 2000.
[31] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking," Journal of the ACM, vol.50, no.5, pp.752–794, 2003.
[32] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," International Journal on Software Tools for Technology Transfer, vol.9, no.5-6, pp.505–525, 2007.
[33] T. Ball, V. Levin, and S.K. Rajamani, "A decade of software model checking with SLAM," Communications of the ACM, vol.54, no.7, pp.68–76, 2011.

Masahiro Matsubara received the B.S. degree in System Information Engineering from the University of Tokyo in 2001. He is currently with Hitachi Automotive Systems, Ltd.

Tatsuhiro Tsuchiya is a professor at the graduate school of information science and technology at Osaka University. He received his M.E. and Ph.D. degrees from Osaka University in 1995 and 1998, respectively.