A practical model-checking (MC) approach for fault analysis, that is one of the most cost-effective tasks in software development, is proposed. The proposed approach is based on a technique, named “Program-oriented Modeling” (POM) for extracting a model from source code. The framework of model extraction by POM provides configurable abstraction based on user-defined transformation rules, and it supports trial-and-error model extraction. An environment for MC called POM/MC was also built. POM/MC analyzes C source code to extract Promela models used for the SPIN model checker. It was applied to an industrial software system to evaluate the efficiency of the configurable model extraction by POM for fault analysis. Moreover, it was shown that the proposed MC approach can reduce the effort involved in analyzing software faults by MC.

1. Introduction

Model checking (MC) may be useful for software development. In the case of developing systems that have concurrency or nondeterminism, some execution paths can rarely be executed because of the low probability of reaching them. Any defects in such rarely reached paths are difficult to find and fix. The huge amount of effort needed in verifying such paths is an issue in regard to practical software development.

MC is a strong candidate to solve this issue. In a previous study, we applied MC to fault analysis, which detects the fault that causes a failure observed in testing or operation[1]. In the case of fault analysis, a model (which represents the behavior of the target system) is constructed by a developer, and negation of the characteristics of the failure (which is also defined by the developer according to the failure observed in testing or operation) is regarded as a desired property for the target system. MC then finds a state in which the property is violated and generates a counterexample, which is a path to reach the state.

However, MC is not frequently used in practical software development, since using it in practice faces several considerable issues. One issue is a problem called “state-explosion,” which prevents application of MC to full-scale software. To avoid state explosion, the target software should be degenerated by extracting a model, and the model has to keep the characteristics to be checked. Inadequate model-extraction sometimes results in failing to obtain paths containing the fault or mistakenly obtaining spurious counterexamples that does not occur on the original software. To construct a proper model, iterative model-extraction in a trial-and-error manner is thus required. Our experiences of fault analysis using MC show that a configurable model-extraction technology, namely, trial-and-error model extraction that avoids both state explosion and spurious counterexamples, is useful for practical software development.

In this paper, a model-checking approach for fault analysis based on a configurable model-extraction framework, named “program-oriented modeling” (POM), and a source-code MC environment, named “POM/MC”[2] are proposed. POM is a conceptual framework for analyzing the source code of target software and extracting the software model that users require. POM/MC analyzes a C source code and extracts Promela models for the SPIN model checker[3]. The rule-based model extraction from the target software enables the configurable model extraction for MC in a trial-and-error manner. In addition, it evaluates the flexibility of model extraction of the POM/MC and the effects of the effort involved in fault analysis with MC, two case studies (using a small but non-trivial toy program and a practical industrial system) were conducted.

The main contribution of this work is a MC approach for fault analysis based on a configurable model extractor. In particular, a model extraction tool using Eclipse Modeling Tools[4], was built, and case studies including a fault analysis using a real industrial software system were conducted. The results of the case studies indicate that the proposed approach reduces the effort for fault analysis.

The remainder of this paper is organized as follows. The motivating example is described in Sect. 2, and the basic concept of using MC for fault analysis is explained in Sect. 3. The POM approach is proposed in Sect. 4, and the POM/MC tool is proposed in Sect. 5. Case studies are described in Sect. 6, and the results of the studies are presented in Sect. 7. Related works are presented in Sect. 8, and conclusions are given in Sect. 9.

2. Motivating Example

Part of the source code used in the first case study presented in Sect. 6 is shown in Fig. 1. The source code represents a mock seat reservation system, which is a client-server system with message communication. In this scenario, a devel-
oper constructs a verification model in the Promela language using the C source code as a reference.

Basically, the developer tries to translate the C language into the Promela language. However, the developer cannot obtain a complete Promela model using only literal translation. The developer should identify the program elements that are mapped to the Promela's built-in concurrency facilities, such as process generation and message passing.

Such identification requires domain- or application-specific knowledge. For example, the `run` statement is translated from the `pthread_create` function, which is a Pthreads library function. The `terminal_thread` function is also translated into proctype, because the function pointer of `terminal_thread` is passed to `pthread_create`, i.e., `terminal_thread` is ‘run’ as a thread. These translations are performed by using the knowledge that this program uses the Pthreads library. On the other hand, translation from `chan_send` into Promela’s message sending statement is an application-specific translation because the function is used only in this program.

After performing the translation and mapping as described above, the developer can run the MC. However, to finish the MC, the developer should abstract the model in a manner that avoids state explosion. Figure 1 illustrates a simple abstraction, where the number of running terminals is reduced from 10 to 3. To reduce the state space enough to finish the checking, the developer typically requires the combination of several abstractions through a trial-and-error process.

This example scenario reveals that a configuration dedicated to the target software or specific verification is required to extract a ‘model-checking-capable’ model, although the model extraction is basically performed by almost literal translation.

### 3. Model Checking and Fault Analysis

#### 3.1 Model Checking

Model checking (MC) is a technique that exhaustively explores the behavior of a system and verifies whether any paths satisfy the given property. The SPIN model checker used in the proposed approach, is a model checker based on the theory of ω-automata (Büchi automata). SPIN accepts the model of the target system \( P \) written in Promela and the property formulated in LTL formula \( f \). It then converts \( P \) to a corresponding Büchi automaton, \( S(\cdot) \). Satisfiability of \( f \) with \( P \) is interpreted as the following problem: whether \( f \) holds for an ω-run \( \sigma \) of \( S(\cdot) \); that is, \( \sigma \models f \). An ω-run for which \( S(\cdot) \) does not hold is output as a counterexample. (SPIN actually converts both \( P \) and \( f \) to Büchi automata.)

In this paper, \( P, \varphi \) denotes a counterexample of property \( \varphi \) on Promela code \( P \); that is, \( P, \varphi \not\models \varphi \).

#### 3.2 Fault Analysis Using MC

##### 3.2.1 Fault Analysis

The terms fault and failure are defined as follows: A fault is a manifestation of an error in software and a failure is termination of the ability of a program to perform a required function or its inability to perform within previously specified limits [5]. The activity for detecting and locating the fault that caused an observed failure is called fault analysis.

During practical software development, developers try to reproduce such failures to analyze the fault to recognize what was happening on the target software when the failure was observed. The basic concepts and difficulties concerning fault analysis are shown in Fig. 2 (a) and (b). When a failure is observed in operation, intermediate states between the initial state and failure are often unobservable (Fig. 2 (a)). This means that developers cannot grasp what caused the failure. In such cases, they execute the program again to reproduce the failure (Fig. 2 (b)). However, reproducing a failure by execution sometimes fails because of the low probability of the fault re-occurring.

##### 3.2.2 Fault Analysis Using MC

The concept of fault analysis using MC is illustrated in
of the negation of the property, i.e., a failure is defined. Next, the program is checked in terms of the property that specifies a characteristic of the target system. We then run the SPIN model checker using the source code and documents of the target system. The model-checking tool checks property \( \phi \), which represents the exact semantics of the source code. It is then translated into an abstracted model that does not depend on any specific program language such as C language. To obtain a model that suits the intention of the user, the transformation rules configured by the user are applied to the abstracted model.

4.2 POM Conceptual Framework

Figure 3 illustrates the POM conceptual framework, which consists of a parser, a model-transformation engine, and a target-code generator. Three types of intermediate models are defined as follows:

**Implementation model** A model that represents a program structure that can be directly acquired by analyzing a source code.

**Abstract program model** A model that acts as a bridge between the implementation model and target model, and plays a role as an abstraction target.

**Target model** A model that is an internal representation of the target code, which is provided to the user. It is translated to a target code according to a model-to-text template.

To extract a model from the source code, first, the parser extracts the implementation model from the source code. The model-transformation engine then translates the extracted implementation model into the abstract-program model and applies the abstractions given by the user to the code, another model, i.e., \( P' \), is built on the basis of the fixed source code, to examine whether the fault is actually fixed, and the model checker checks property \( \phi \).

In each case, we successfully reproduced the failure with its trace as a counterexample within one to ten weeks. During that time, it was found that developing a model that is sufficiently precise to reproduce the target failure and sufficiently abstracted to avoid the state explosion is an error-prone and time-consuming task. Constructing the models requires too much effort to use MC casually. A trial-and-error procedure for simulating system’s behavior and avoiding state explosion, by extracting models automatically on the basis of rules describing model abstractions, was therefore focused on in this work.

3.2.3 Lessons Learned through Experiences

We applied MC to fault analysis, especially to failure reproduction, at several software-development sites. When a failure is observed in testing, a verification model, \( P \), is described in Promela by using the source code and documents of the target system. We then run the SPIN model checker using \( P \) and a checking property, i.e., \( \varphi \) which represents that “the target failure does not occur”. Eventually SPIN outputs an “invalid” result with counterexample \( P', \varphi \), which represents the execution trace of the fault that causes the target failure. In addition, when the fault is fixed on the source code, another model, i.e., \( P' \), is built on the basis of the fixed source code, to examine whether the fault is actually fixed, and the model checker checks property \( \varphi \).

In each case, we successfully reproduced the failure with its trace as a counterexample within one to ten weeks. During that time, it was found that developing a model that is sufficiently precise to reproduce the target failure and sufficiently abstracted to avoid the state explosion is an error-prone and time-consuming task. Constructing the models requires too much effort to use MC casually. A trial-and-error procedure for simulating system’s behavior and avoiding state explosion, by extracting models automatically on the basis of rules describing model abstractions, was therefore focused on in this work.

4. Program-Oriented Modeling

The POM approach is described as follows. Herein, the term “model” is defined as “a representation of a system from a specific viewpoint with a specific abstraction.”

4.1 Approach

The basic concept of model extraction by POM is a stepwise model transformation according to the rules configured by a user. The initial model, called the “implementation model,” represents the exact semantics of the source code. It is then translated into an abstracted model that does not depend on any specific program language such as C language. To obtain a model that suits the intention of the user, the transformation rules configured by the user are applied to the abstracted model.

4.2 POM Conceptual Framework

Figure 3 illustrates the POM conceptual framework, which generates a target code (namely, a textual representation of the target model extracted from a given source code). It consists of a parser, a model-transformation engine, and a target-code generator. Three types of intermediate models are defined as follows:

**Implementation model** A model that represents a program structure that can be directly acquired by analyzing a source code.

**Abstract program model** A model that acts as a bridge between the implementation model and target model, and plays a role as an abstraction target.

**Target model** A model that is an internal representation of the target code, which is provided to the user. It is translated to a target code according to a model-to-text template.

To extract a model from the source code, first, the parser extracts the implementation model from the source code. The model-transformation engine then translates the extracted implementation model into the abstract-program model and applies the abstractions given by the user to the
intermediate model. The target model is then translated from the abstract-program model. Finally, according to the template, the target code generator writes the target code corresponding to the target model.

Each intermediate model type has a corresponding metamodel defined on the basis of MOF technology [6]. The transformation engine transforms one model into another model repetitively. These transformations are scripted in the QVT model transformation language [7].

\( r(m) \) herein denotes the result of transformation of model \( m \) according to transformation rule \( r \). If \( R \) is an ordered set of transformation rules consisting of \( n \) rules from \( r_0 \) to \( r_{n-1} \), \( R(m) \) denotes the result of repetitive transformation of \( m \) according to \( R \), such that \( R(m) = r_{n-1}(\ldots(r_0(m))\ldots) \).

\( c \) is taken as a source code, and \( R \) is taken as a transformation rule that is given to the POM framework. \( R \) is tuple \( \langle R_{i\rightarrow a} , R_{a\rightarrow c}, R_{c\rightarrow t} \rangle \), where \( R_{i\rightarrow a} \) is an ordered set of rules for transformation from an implementation model to an abstract-program model, \( R_{a\rightarrow c} \) is an ordered set of rules for transformation from an abstract-program model to another abstract-program model, and \( R_{c\rightarrow t} \) is a ordered set of rules for transformation from an abstract-program model to a target model. \( c \) is transformed to implementation model \( im \) by the parser. First, the initial abstract-program model is transformed from \( im \) according to \( R_{i\rightarrow a} \), that is \( R_{i\rightarrow a}(im) \). Next, the initial abstract model is transformed to \( R_{a\rightarrow c}(R_{i\rightarrow a}(im)) \). That transformation gives target model \( R_{c\rightarrow t}(R_{a\rightarrow c}(R_{i\rightarrow a}(im))) \). Lastly, according to the template, the target code is generated from the target model by the target-code generator.

The parser and \( R_{i\rightarrow a} \) are defined for the “source language” in which the source code is written. The template for the target-code generator and \( R_{i\rightarrow a} \) is also defined for the “target language” in which the target code is written. Only \( R_{a\rightarrow c} \) has the role of deciding the content of the target code that is provided to the user. Therefore, when the source language and the target language are specified, transformation from \( c \) to the target code on the POM framework can be characterized only by \( R_{a\rightarrow c} \). Hereafter, a result of transformation of a source code \( c \) in a source language \( L_i \) into a target language \( L_t \) according to \( R_{a\rightarrow c} \) on the POM framework is denoted as \( POM_{L_i\rightarrow L_t}(c, R_{a\rightarrow c}) \).

5. POM/MC: POM for MC

POM/MC, which is a POM application that extracts Promela models from C source code, is described in the following.

5.1 POM/MC

The POM/MC environment extracts the Promela code from a source code, and checks the Promela code against the property that the user defined. \( c \) is taken as a source code, and \( R \) is taken as a transformation rule. \( R_{i\rightarrow a} \) is defined for the “source language” in which the source code is written. \( R_{a\rightarrow c} \) is defined for translation of the C language and \( R_{c\rightarrow t} \) is defined for translation of abstract-program model to the Promela language. The target code, \( POM\langle c, R_{a\rightarrow c} \rangle \), is generated by the POM framework and is checked against the property \( \varphi \) by SPIN. If \( \varphi \) is not satisfied \( \varphi \), a counter example, \( POM\langle C\rightarrow Promela\langle c, R_{a\rightarrow c} \rangle, \varphi \rangle \), is generated.

5.2 Extraction of Promela Model from C Source Code

Figure 4 shows the basic idea of the process of model extraction with abstraction (Note that here the models are represented as code for ease of reading). Model-to-model transformation is basically performed in a literal transformation manner, e.g., a C for block is translated into a Promela do block (Fig. 4 (a)). Meanwhile, transformation from the function call of pthread_create into the run statement is a domain-specific transformation that includes multiple program elements. Such a transformation is defined by the rules configured by a user. The transformation from pthread_create into run comprises the following two steps (Fig. 4 (b)). In the first step, the rule for the Pthreads library identifies the pthread_create function call as the “process creation statement.” The identified function call is transformed to an abstract element that represents a function call with the “process creation” role. In addition, the rule transforms the model so that the function definition, which is specified as an argument of the pthread_create function, plays the role of a “process.” The model-to-model transformation rule that transforms an abstract-program model into a Promela model then transforms the abstract element that represents “process creation” into a run statement.
5.2.1 ASG-Based Metamodel

Figure 5 presents the metamodels defined for POM/MC. (Note that these metamodels are simplified to describe the concept more clearly.) Basically, each metamodel has a structure based on an abstract semantic graph (ASG) [8], which is an abstract syntax tree with the reference (i.e., name-binding) information. The reference information is represented by the Ref (reference) class. The abstract-program metamodel is designed to represent a procedural language generally based on the C language metamodel. Characteristic elements of the abstract-program metamodel are logical element classes, which play a key role in achieving model abstraction. By transforming an element into a logical element, a specific role can be assigned to the element.

5.2.2 Model Abstraction by Multi-Phased Model Transformation

The transformation process by which to acquire a Promela model from a C model is accomplished by applying a set of transformation rules configured by the user to the intermediate models. This process is comprised of the following three steps.

(1) C → Abstract Program

The first step of the model transformation is an inter-model transformation from a model conforming to the C metamodel into a model conforming to the abstract-program metamodel. This transformation is an approximately one-
to-one element mapping because the structure of the abstract program basically conforms to the structure of the C metamodel.

(2) Abstraction

 The second step is in-model transformation of the abstract-program model that implements model abstraction. This step is comprised of the following phases.

1. Syntax normalization: Prior to the other abstractions, syntax variations are unified. For example, the nested assignment is expanded from, for example, \( a = (b = c) \) to \( a = b; c; a = b \).

2. Environmental mapping: As the main part of the abstraction, library functions/variables that interface with the external environment (e.g., hardware) or utilities are identified and mapped to stubs or logical elements.

3. Reduction: Avoiding state explosion is the aim of this transformation, such as removing unnecessary procedures and abstraction of data structures.

4. Semantics adaptation: Finally, the elements that do not have any exactly corresponding elements in the Promela metamodel are transformed into an equivalent structure that can be directly transformed into the Promela model. For example, a multi-dimensional array is transformed to a representation using a single-dimensional array and a structure element because the Promela language only supports a single-dimensional array.

(3) Abstract program → Promela

 The final step of the model transformation is an inter-model transformation from a model conforming to the abstract program metamodel to a model conforming into the Promela metamodel.

 Examples of the model-transformation rules constructed for POM/MC previously by the authors are listed in Table 1. The Promela language does not have enough expressiveness to support all of the features of the C language; for example, it cannot be used to fully simulate pointer operations. In contrast, POM/MC analyzes the pointer references and transform pointers to actual references by the transformation rule (“Alias analysis of pointer” rule in Table 1). This transformation is not sound in general but practical under the assumption that references of pointers can be detected by static analysis. When this assumption is not supported, to simulate pointer operations, the user selects/constructs other transformation rules under other assumptions.

### Table 1  Example of model transformation rules.

| Abstraction Type | Name | Description |
|------------------|------|-------------|
| **Semantics Conversion** | Nest unfolding of expression | Unfold expressions in C language such as substitutions and function calls, which are statements in Promela, e.g. \( a = (b = c) \) → \( b = c; a = b \). |
| | Decomposition of multi-dimensional array | Decompose the multi-dimensional array in C language to the structure field having an array as a factor for one dimension only. |
| | Alias analysis of pointer | Analyze the change in pointer reference, transform the dereference pointer operation to an actual reference, e.g., \( *i \) to \( \text{int} i; *j = j; a = b \). |
| | Phreads | Convert the mechanism offered in the POSIX thread libraries (Phreads) such as thread generation and exclusive control into a mechanism of a process and the guard of Promela. |
| | Entry point | Convert a function specified in the parameter of this rule into the initial process (init) of Promela. |
| | State machine driver | Convert a function specified in the parameter into the structure to be periodically called, and reproduce a state transition design. |
| | Stub | Convert a specified function into its stub. Four types of stubs are currently supported: stubs that (1) have no function, (2) return a constant value, (3) return random numbers of the designated range, and (4) call the modeled code defined separately. |
| **Abstraction** | Optimization of type declaration | According to variables and functions declared with the integer type, maximum values and minimum values are obtained from substitution relations. If the smaller type is enough to represent the range, change the type to a smaller one. |
| | Deletion of unused declarations | Demand a declaration (argument, function, and type) that may be used recursively from a driver and delete the declaration that is not used. This rule is applied after an outside-world model is constructed. |

LLVM/Clang\(^7\) is used as a C parser. As a model-transformation platform, Eclipse Modeling Tools (EMT) [4], which is an Eclipse distribution for building a modeling environment on the basis of the Eclipse Modeling Framework (EMF) [9], is used. The metamodels are defined as the Ecore model and code rules using the Operational QVT (QVTo) plug-in\(^8\), which is an EMF implementation of MOF QVT/OperationalMapping. The target code is written out using Acceleo\(^9\), which is an implementation of M2T template [10].

QVTo is a language that was originally developed for inter-model transformation and model refinement of the model-driven architecture process. Figure 6 is a simplified sample of a part of the “Phreads” rule in Table 1. This rule maps each function call related to the Phreads library into its corresponding element defined by the metamodel for the abstract-program model. Declarations of the function call are also mapped into their corresponding elements. For example, the call of function “pthread_create” is mapped

---

\(^7\)http://llvm.org, 2012-06-01
\(^8\)http://www.eclipse.org
\(^9\)http://www.acceleo.org, 2012-06-01
to an element called “ProcessRunningFunctionReference,” which is a reference of a function that runs as a process.

QVTo has the capability to define more complex transformation algorithms. The “Optimization of type declaration” rule, for example, automatically traces the abstract-program model and recognizes constant values and their influence on variables from substitution relations, in order to detect the maximum values and minimal values that are possibly assigned to a user-specified variable. The rule can therefore reduce the range of the variable.

6. Case Studies

Case studies were conducted to investigate the following two questions:

- Question 1. *Can POM/MC extract a model-checking-capable Promela model from a C source code with domain- and application-specific abstraction?*
- Question 2. *Can POM/MC reduce the effort required for fault analysis using MC in the case of industrial-scale software?*

6.1 Seat Reservation Problem

To investigate Question 1, a C program that simulates a simple seat reservation system, by which several clients send “checking availability,” “reservation,” and/or “confirmation” queries to a server, was prepared. This program contains a double-booking fault, where the server provides a response of “available” to multiple clients that request the same seat simultaneously.

The Pthreads library, which is an implementation of the POSIX thread, was used to implement concurrent behaviors of the client-server system. A server and clients run on their own threads. In addition, the source code contains a printf call to display the status of the system, which should be removed before MC is performed. The LOC of the code is approximately 500.

6.1.1 Rule Configuration

To use the Promela functionalities, rules for environmental mapping need to be formulated, and to avoid state explosion, reduction rules need to be formulated. A rule set that maps the use of the functions and the types of the Pthreads library to the concurrent grammar of the Promela language was prepared. Note that this rule set is applicable to not only this case study but also any code using the Pthreads library.

Figure 7 shows the abstraction flow. The original C source code, the corresponding Promela model, and abstract-program models that present the model transformation are illustrated in the figure (other intermediate models are omitted). The pthread_create function is mapped to the run statement in the Promela model, and the function declaration that is specified as the target function is mapped to the proctype declaration. The pthread_create function is transformed into the run statement by replacing the FunctionRef object related to the FunctionCall object of the pthread_create function with the ProcessRunningFunctionRef object. A function with a ProcessRunningFunctionRef object is mapped to an element of the run statement of the Promela model with the model-to-model transformation rule. Therefore, we code to generate ProcessRunningFunctionRef objects in the abstract-program model.

By using the result of the run statement, i.e., the generated ProcessRunningFunctionRef object, the target function of pthread_create is mapped to the proctype declaration. Function declarations that are referenced by the ProcessRunningFunctionRef objects are mapped to a ProcessFunctionDecl object. Similar to the ProcessRunningFunctionRef object, the ProcessFunctionDecl object is mapped to an element of the proctype declaration of the Promela model according to the model-to-model transformation rule.

The following reduction rules, for avoiding state explosion in this case study were prepared.

- Replace the value of the constant variable that represents the number of seats.
- Replace the loop block that repeats the entire process (i.e., initialize and repeat the seat-reservation process if all of the seats are reserved) with a non-loop block.

6.1.2 Results

The rules described above were used to extract a Promela model (with LOC of approximately 700; see Table 2 for a summary), and an error (i.e., reproduced the target fault) was successfully detected using assertion checking.

6.2 Industrial Embedded Software

To answer Question 2, a case study of MC using POM/MC in a fault analysis scenario concerning an actual industrial
software system was conducted. The target software is an embedded software system that controls a conveyor system comprised of multiple hardware modules. The hardware modules communicate with each other via a network to convey an object from one module to another. This software system is built on a message-driven state machine architecture and has dedicated libraries for network communication and hardware manipulation.

In this case study, we analyze a dead-lock fault that occurs during the handshake of the inter-module conveying caused by an unexpected message sequence. The source code of the conveying control is approximately 13,000 LOC.

6.2.1 Experimental Settings

The tasks of the participants working on the fault-analysis are listed in Table 3. The first participant had worked on QVTo implementation for three months. He attended a one-day lecture on the architecture and conveying specification of the target system used for this case study. The second participant had been involved in the development of the target system and understood the overall specifications and design of the target system. Both participants had experience of MC. The first participant primarily designs and writes the rules, and the second participant primarily performs MC and directs abstraction design.

In this case study, it was assumed that the following information concerning the fault is known as the minimum information required for model checking:

- The fault is a dead-lock fault, namely the internal state transition stalls
- It occurs when a module receives a specific signal during the hand-shaking of inter-module conveyance.

6.2.2 Fault Analysis Scenario

Figure 8 depicts the procedure of the fault analysis.

1. **Designing the model to be extracted**: Specify the range of source code to be extracted as a model and abstract its boundary to the external environment. For example, replace all hardware manipulation functions with empty stubs or represent the inter-module communication as read/write operations of global variables.
2. **Configuring rules**: In accordance with the model designed above, choose general (built-in) rules or write rules dedicated to this problem (or this software).
3. **Describing the fault as a checking property**: In this case, the property is specified with the LTL formula that indicates that a specific state is always reachable.
from another state.

4. **Extracting models using POM/MC**: POM/MC takes the source code and the rules configured in step (2) as input and automatically generates a Promela model.

5. **MC**: Run SPIN to determine whether the model generated in step (4) satisfies the property specified in step (3).

6. **Evaluating the MC result**: Evaluate the MC result to determine whether the target fault is detected. If SPIN detects an error and the counterexample trace of the error represents the target fault, this analysis process finishes with the execution trace of the target fault. Otherwise, if the trace of the detected error does not appear in the original source code or the MC fails because of the state explosion, return to step (1) or step (2) and retry. We judged that a “state explosion has occurred” when the model checker does not finish within two hours.

### 6.2.3 Results

We successfully detected a target fault after eight rule configurations, as shown in Table 4; i.e., the participants repeated the trial-and-error loop of the fault analysis process seven times.

When MC resulted in state explosion, the participants abstracted the target model by reconfiguring the transformation rules to reduce the state space of the target model. In step 3 in Table 4, for example, they bypassed a process for handling the parameters in order to reduce the number of states for interaction between one process and another. However, the counterexample that the model checker generated was a spurious one. When MC resulted in a spurious counterexample, the participants unfastened abstraction or tried other ways of abstraction. In step 4, they gave up the “bypass parameter process” abstraction and tried another abstraction method, namely, one that simplified networking. Step 4 resulted in another state explosion, and they further simplified the network in step 5. The abstraction in the trial-and-error manner was performed in these ways until a fault was detected in step 8.

The above-described fault analysis takes approximately four days (seven man-days), excluding the initial lecture and survey of the target system and executions of POM/MC and SPIN. Note that additional inspection of code and documents (to examine the spurious counterexample) is included. The final model has approximately 2,000 LOC (see Table 2).

### 7. Discussion

#### 7.1 Applicability of MC to Fault Analysis

Specification documents are sometimes fragmentary and abstract since they are developed from the viewpoint of development of a target system. It is not realistic, in regards to cost, for developers to write specification detailed enough to be verified. Constructing a behavioral model of the target system has been an issue in regard to using MC in practice. In the case of the proposed approach to MC, the source code of the target system is transformed into a verification model with transformation rules. In other words, a model for verification does not have to be prepared by developers.

Exhaustively enumerating properties, which represent desirable/undesirable states, transitions or combinations of them, is another issue concerning using MC. This is a similar problem as test-case design in the research field of software testing. For example, Ogawa et al. proposed an approach for enumerating the properties for MC on the basis of goal-oriented requirement analysis [11]. In that case, the desired properties at requirement or specification level are obtained as goals; however, implementation-level properties that have to be satisfied on software cannot be specified. Although MC is expected to be applied to finding unknown defects, it is still difficult to be practically applied because of the issue concerning exhaustiveness of properties. In a fault-analysis scenario, the failure is observed and can be defined as an undesirable property. Accordingly, application of MC to fault analysis is a more practical scenario than its application to verification of systems.

Configurable model extraction can support trial-and-error failure reproduction; however, detecting the fault that caused the observed failure remains another issue. One approach to detect the fault is to analyze the counterexamples obtained by MC. Counterexamples that represent the paths reaching the failure states may have much information to characterize the fault. Another approach to detect the fault is to narrow the verification model in order to specify the conditions that cause the failure. In fault analysis by execution of software, developers often modify the target system in order to satisfy the condition under which they assumes the failure will occur. We believe that the hypothesis-based configuration of model-extraction rules will be able to enable fault detection as practical debugging [1].

#### 7.2 Rule Configuration in a Fault Analysis Scenario

In the second case study (i.e., using a conveyor system) the fault was successfully reproduced in seven man-days which is much shorter than previously experienced with MC using hand-coded models. In a similar case (similar code base
and extracted model sizes, it took one man-month (20 man-days, excluding initial survey and SPIN execution) to detect the target fault by MC.

The abstractions used in the conveyor system case are simpler than those used in the hand-coding case (which requires a similar amount of state reduction). In the hand-coding case, more than 20 abstractions, including complex abstractions such as simplification of logics and modification of data structures, were applied. On the other hand, in the conveyor case, approximately 10 abstractions comprising simple abstractions, such as function stubs, were used. Thus, the effort involved in not only eliminating the writing cost but also simplifying in the abstraction configuration was reduced.

This difference in the abstraction configuration between the second case study and the hand-coding case likely comes from the following two factors. The first is the difference in the software architecture between the cases. The software architecture of the conveyor system has a unified interface for low-level operations, such as hardware manipulation, so that stubs for simulation testing can be more easily created. The interface functions were simply replaced with stubs using parameterized rules for stubs.

The second factor is automation of model extraction. When coding models from scratch by hand, using the trial-and-error process of MC, the modeler tends to begin with a small model with core functionalities and processes flows and then add details as needed. This method is considered to be abstraction, where multiple abstractions, including complex abstractions, are applied first, and then abstractions are removed in a trial-and-error manner. On the other hand, the abstraction process with automated extraction using rules begins MC with an implementation-level model and abstracts it in small steps as needed. This process promotes proper abstraction, by first using effective but simple abstraction, followed by complex abstraction as a last resort.

7.3 Threats to Validity

Internal validity: The effort involved in fault analysis with MC was evaluated on the basis of a comparison between cases, in which effort is measured informally, involving different software systems. However, the case study reported here has confirmable properties in terms of source LOC, model LOC, and ability/knowledge level of a modeler. External validity: As discussed above, rule configuration depends on the architecture of the target software system. If a target system conforms to some rules defined as the architecture, these rules can be coded as QVT rules. However, it may be difficult to configure rules for the legacy system in which the architecture decays by ad-hoc fixes.

8. Related Works

Several approaches to extract Promela models from source code have been proposed. Modex (FeaVer) [12] extracts models from the C source code of a specific platform. Modex first separates a source code into a target code and an external (e.g., framework) code; the target code is then translated into a Promela model by using a “translation map,” in which users specify abstractions; after that Modex combines the obtained model with an external model.

This approach, based on translation, has the following problems in regard to specifying abstractions. To configure abstractions with this approach, users should compose syntax-level C to a Promela translation map by synthesizing multiple abstractions in consideration of their interference. This is an error-prone task and requires considerable efforts, especially in a fault-analysis scenario (which typically requires trial-and-error configuration of abstractions). In addition, a translation map only supports syntax-level patterns. For example, to specify abstraction of the Pthreads library (presented at Sect. 6.1.1), users should identify all Pthreads-related program constructs, such as the “thread” functions (terminal_thread in Fig. 7), and enumerate them in the map.

Another approach is the compiler approach. Bandera [13] is a model extractor created from Java source code that supports multiple types of output including Promela. Bandera uses Jimple intermediate representation, which is originally for compiler optimization, to optimize models. Using control/data flow and program dependence in Jimple, Bandera applies program slicing and data abstraction (e.g., equivalence partitioning), which is configured by the user.

This approach supports effective and conservative abstraction; meanwhile, users can not define drastic abstraction in which the program structure is modified destructively, which is often required in fault-analysis scenarios.

Androutsopoulos et al. proposed a model abstraction method for a state machine model [14]. Their method automatically identifies and removes the model elements that are not referenced transitively by specifying states or events. Although POM/MC also supports this kind of model abstraction, it only uses simple reduction rules based on ASG. By analyzing the behavior, such as data flow, in greater detail, we are able to abstract more effectively.

9. Conclusion

A model-checking (MC) approach for fault analysis based on the “program-oriented modeling” (POM), which is a rule-based approach to extract software models from source code, was proposed and evaluated. The POM framework provides a configurable abstraction that is applied to a model according to user-defined transformation rules. POM/MC, which extracts a Promela model from C source code for MC, was also developed. The results of case studies using the source code of an actual industrial system indicate that POM/MC reduces the effort required to construct models for MC in fault-analysis scenarios. In the case of an industrial system, the fault analysis was completed in seven man-days, whereas it takes 20 man-days to complete by manual modeling. The results of the case studies mean that MC with configurable model extraction enables fault analysis in
a trial-and-error manner.

References

[1] H. Ogawa, M. Ichii, F. Kumeno, and T. Aoki, “A practical study of debugging using model checking,” Proc. Asia-Pacific Software Engineering Conf. (APSEC), pp.134–139, 2013.

[2] M. Ichii, T. Myojin, Y. Nakagawa, M. Chikahisa, and H. Ogawa, “A rule-based automated approach for extracting models from source code,” Proc. Working Conf. on Reverse Engineering (WCRE), pp.308–317, 2012.

[3] G.J. Holzmann, The spin model checker: Primer and reference manual, ISBN: 0-321-22862-6, 2004.

[4] R.C. Gronback, Eclipse modeling project: A domain-specific language (DSL) toolkit, Pearson Education, 2009.

[5] ISO/IEC/IEEE. ISO/IEC/IEEE24765:2010 Systems and software engineering - Vocabulary, 2010.

[6] Object Management Group, “Meta object facility (MOF) core specification.” [Online]. Available: http://www.omg.org/spec/MOF/2.0/ PDF, 2006.

[7] Object Management Group, “Meta object facility (MOF) 2.0 query/view/transformation.” [Online]. Available: http://www.omg. org/spec/QVT/1.0/PDF, 2008.

[8] D. Binkley, “Source code analysis: A road map,” in Future of Software Engineering, pp.104–119, IEEE Computer Society, 2007.

[9] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.

[10] Object Management Group, “MOF model to text transformation language, v1.0.” [Online]. Available: http://www.omg.org/spec/MOFRM2T/1.0/PDF, 2008.

[11] H. Ogawa, F. Kumeno, and S. Honiden, “Model checking process with goal oriented requirements analysis,” Proc. Asia-Pacific Software Engineering Conf. (APSEC), pp.377–384, 2008.

[12] G.J. Holzmann and M.H. Smith, “An automated verification method for distributed systems software based on model extraction,” IEEE Trans. Softw. Eng., vol.28, no.4, pp.364–377, 2002.

[13] J.C. Corbett, M.B. Dwyer, J. Hatchill, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng, “Bandera: Extracting finite-state models from java source code,” Proc. Int. Conf. on Software Engineering (ICSE), pp.439–448, 2000.

[14] K. Androuloupoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li, “Model projection: simplifying models in response to restricting the environment,” Proc. Int. Conf. on Software Engineering (ICSE), pp.291–300, 2011.

Hideto Ogawa received his M.S. degrees in information engineering from Nagoya University in 1996. He is a senior researcher at Yokohama Research Laboratory, Hitachi, Ltd. His research interests include formal methods, software testing and software development processes. He is a member of the IEICE and IPSJ.

Makoto Ichii received his M.S. and Ph.D. degrees in information science and technology from Osaka University in 2006 and 2009, respectively. He is a researcher at Yokohama Research Laboratory, Hitachi, Ltd. His research interests include program analysis, formal methods and model-driven engineering techniques. He is a member of the IEEE and IPSJ.

Tomoyuki Myojin received his B.E. degree from University of Fuku in 2004 and his M.S. degree from the University of Electro-Communications in 2006. He is a researcher at Yokohama Research Laboratory, Hitachi, Ltd. His research interests include formal specification and verification techniques for embedded systems development.

Masaki Chikahisa received his M.S. degree in information science and technology from Osaka University in 2004. He is a researcher at Yokohama Research Laboratory, Hitachi, Ltd. His research interests include verification techniques for embedded software.

Yuichiro Nakagawa received his B.E. and M.S. degrees in aeronautics and astronautics from Tokai University in 1999 and 2001, respectively. He is a senior researcher at Yokohama Research Laboratory, Hitachi, Ltd. His research interests include software testing techniques. He is a member of the IPSJ.