Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 909762, 21 pages
http://dx.doi.org/10.1155/2014/909762

Research Article

Formal Specification Based Automatic Test Generation for Embedded Network Systems

Eun Hye Choi,[1] Hideaki Nishihara,[1] Takahiro Ando,[2] Nguyen Van Tang,[3] Masahiro Aoki,[4] Keiichi Yoshisaka,[4] Osamu Mizuno,[5] and Hitoshi Ohsaki[1]

[1] *National Institute of Advanced Industrial Science and Technology, Japan*
[2] *Kyushu University, Japan*
[3] *Hanoi University of Industry, Vietnam*
[4] *Daikin Industries, Ltd., Japan*
[5] *Kyoto Institute of Technology, Japan*

Correspondence should be addressed to Eun Hye Choi; e.choi@aist.go.jp

Received 13 September 2013; Revised 22 November 2013; Accepted 24 November 2013; Published 11 May 2014

Academic Editor: Chin-Yu Huang

Embedded systems have become increasingly connected and communicate with each other, forming large-scaled and complicated network systems. To make their design and testing more reliable and robust, this paper proposes a formal specification language called SENS and a SENS-based automatic test generation tool called TGSENS. Our approach is summarized as follows: (1) A user describes requirements of target embedded network systems by logical property-based constraints using SENS. (2) Given SENS specifications, test cases are automatically generated using a SAT-based solver. Filtering mechanisms to select efficient test cases are also available in our tool. (3) In addition, given a testing goal by the user, test sequences are automatically extracted from exhaustive test cases. We've implemented our approach and conducted several experiments on practical case studies. Through the experiments, we confirmed the efficiency of our approach in design and test generation of real embedded air-conditioning network systems.

1. Introduction

An embedded network system (ENS) is a system consisting of a number of embedded units that communicate with each other. Nowadays, ENSs are growing rapidly and playing an important role in our daily life, such as in-vehicle networking systems, home appliance networks, and building energy management systems. On the other hand, because of the increasing complexity of functionality and growing network scales, as well as the increasing numbers and types of embedded units in ENSs, development of the ENSs has been becoming more and more difficult. Particularly, design and test generation are ones of the most important and cost-consuming phases in reliable ENS development, but many companies still rely on the traditional techniques where system requirements are described in a natural language and test cases are generated based on human experience. However, design and test by manpower may not sufficiently cover enormous numbers of combination to be considered especially for complicated ENSs.

During the last decade, numerous formal methods have been proposed to improve software quality. Especially, test generation is an attractive application for mechanized formal methods; the importance of good test cases is universally recognized and so is the high cost of generating them by hand. One of these methods, the model-based test generation, has been extensively studied (e.g., see [1–4]). One of the advantages of this technique is that each behavior of the system model is described directly, and then test cases can be automatically generated from the models. Note, however, that if test cases are generated from incomplete specifications and models, they suffer from the problem of lacking and low quality of test cases. Therefore, formal specification based modeling and test generation for large-scale ENSs are still challenges.

This paper presents our works gained in a joint project of AIST, DaiKin Industries, and Kyoto Institue of Technology. The ultimate goal of our project is to develop a robust and reliable automatic test generation tool suitable for a development process of air-conditioning network systems developed by Daikin . (We refer to the company as Company D in the rest of this paper.) The target system consists of a number of embedded controllers and indoor/outdoor air-conditioning equipment for building management. For this project, we have taken an approach of automatic test generation based on a formal specification. In the following, we present three main contributions toward our overall goal.

(i) *Development of a Formal Specification Language*. First, we have developed a specification language for ENSs called SENS (Specification language for Embedded Network Systems). SENS is designed under the concept of test-oriented, object-oriented, modular, and lightweight description of ENSs. In SENS system, the requirements are specified using logical constraints. This logical property rigorously determines a set of state transitions that satisfy all constraints and thus helps to get the specification without lacking and inconsistency and clearly obtain an advantage over a manpower design of system and test.

(ii) *Development of a Test Case Generator*. Second, we have developed a tool called TGSENS (Test Generator based on SENS) which automatically generates high quality test cases from a given specification written in SENS. Our tool can find distinct and enormous test cases quickly that satisfy complex specifications using a SAT solver as a core engine. Our tool can also perform filtering to select efficient test cases. In a feasibility study using air-conditioning systems of Company D, given a specification for a certain function of the system, our tool took only about 12 seconds to generate 77,700 test cases. We expect that our tool will provide both quality improvement and cost saving for testing target ENSs.

(iii) *Development of a Test Sequence Generator*. Third, we have also developed a tool in TGSENS to generate test sequences that satisfy a given testing goal by a user. In an experiment using components of the target air-conditioning systems, our tool generated 156 test sequences in 8 minutes for a given testing goal such that the sequence contains 2 milestone states whose distance is less than 3 state transitions. We confirmed that our tool is helpful for test engineers to design test scenarios efficiently.

The rest of this paper is organized as follows. In the next section, we briefly introduce the overview of our work and our tool. Section 3 describes the features, syntax, and semantics of the SENS language. In Section 4, we propose the method for generating test cases and sequences from SENS specifications. Section 5 presents experimental results when applying our tool to the air-conditioning systems of Company D. Section 6 discusses related works. Finally, we conclude our work and mention future works in Section 7.

2. Overview of Our Work

Our target system is an embedded network system (ENS). An ENS can be considered as a concurrent system consisting of multiple subsystems with global environments, for example, embedded indoor and outdoor equipment and controllers in an air-conditioning network system as illustrated in Figure 1. Each subsystem in an ENS is an event-triggered system whose events are communications with other subsystems and whose operations are constrained by conditions related to its environment and time.

The behaviors of an ENS are specified in a hierarchical way: an ENS consists of subsystems, a subsystem is constructed from its functions, and each function is constructed from a set of functional requirements, as described in Figure 1. To enhance the reliability and efficiency of design and testing ENSs, we first focus on designing and modeling a formal specification language for ENSs, called SENS, which supports the hierarchy of ENSs and is convenient for our test generation purpose. We next achieve an automatic test generation approach based on SENS, called TGSENS. Figure 2 illustrates the overview of our tool that consists of the following 5 steps.

(1) *Specifying Requirements in SENS*. A user describes system requirements of the target ENS using SENS. For the air-conditioning systems of Company D, we have translated original specification documents written in a natural language to formal specifications in SENS.

(2) *Translating SENS Specifications to CNF*. The SENS translator first performs the syntax and type checking for the given SENS specification. Next, the translator automatically transforms logical requirements of the SENS specification to CNF (conjunctive normal form) formulas according to our translation rules.

(3) *Solving CNF Formulas Using a SAT Solver*. The CNF formulas obtained in Step (2) are imported as an input of the solver. The solver automatically finds all possible assignments that satisfy the input CNF formulas by iteratively using a SAT solver.

(4) *Mapping from Assignments to Test Cases*. The set of possible assignments obtained in Step (3) are automatically mapped to the set of state transitions corresponding to one-step test cases.

(5) *Generating Test Sequences under a Testing Goal*. When a user gives a testing goal including remarkable states and lengths of sequences, a set of test sequences are automatically derived from the test cases (state transitions) obtained in Step (4).

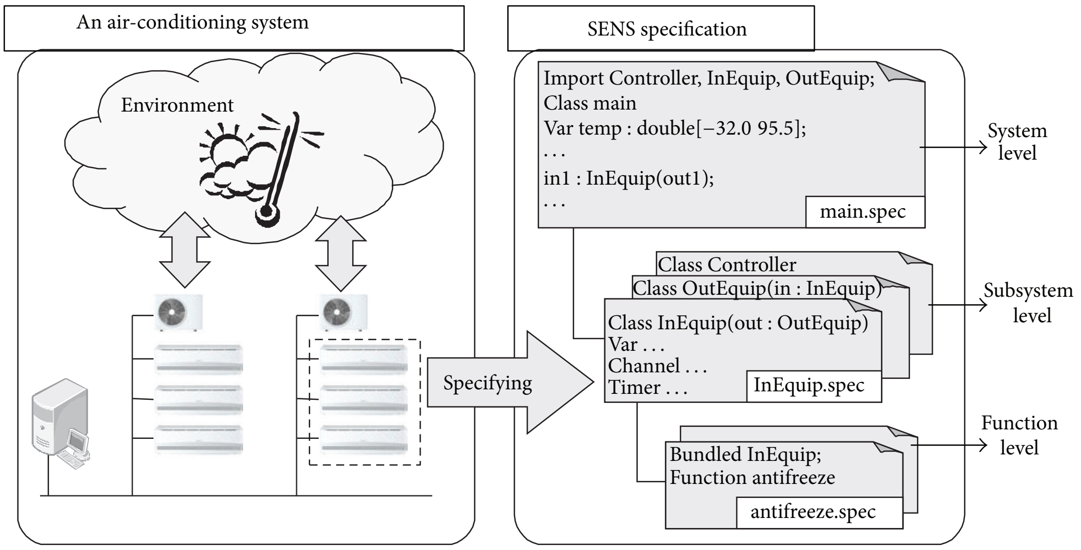


Figure 1: An air-conditioning system and its SENS specification.

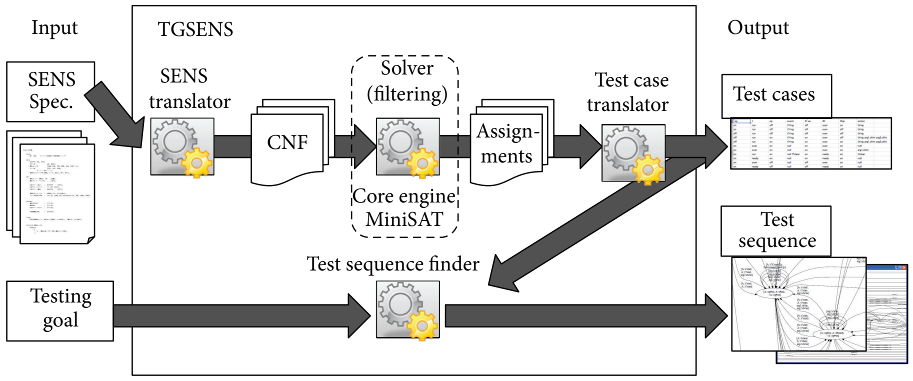


Figure 2: An overview of TGSENS.

3. SENS Specification

In this section, we present a compact formal language called SENS to specify requirements of embedded network systems (ENSs). In Section 3.1, we present features of SENS. In Sections 3.2 and 3.3, we briefly describe the syntax and semantics of SENS.

3.1. Features of SENS. The SENS specification is basically a set of requirements represented by logical formulas. SENS is based on a quantifier-free predicate modal logic with a next operator where predicate variables are classified into environments, systems, communication channels, and timer variables. SENS has the following features.

3.1.1. Property-Based. Property-based logical formulas represent requirements which constrain system behaviors. Since a part of the specification less affects the entire specification, it is easy to modify and reuse SENS specifications. A requirement is described in a traditional Hoare triple-like way:

$$\langle pre : event : (post, action)\rangle, \tag{1}$$

where *pre* (resp., *post*) denotes a logical constraint holding in a prestate (resp., poststate) and an *event* (resp., *action*) represents a message receiving (resp., sending) between subsystems. This 3-tuple requirement needs to hold for all transitions of the system model, and we call this 3-tuple requirement a transition requirement.

3.1.2. Test-Oriented. For a testing purpose, specifying tested values and bounds of variables is available in the variable declaration. This supports a test-oriented design and an efficient test generation from SENS specifications.

3.1.3. Object-Oriented. A group of subsystems is represented as a parameterized class, and each subsystem is defined as an instance of the class in the main class representing the whole system. This representation leads to an efficient description of a large number of subsystems in an ENS.

3.1.4. Modular. Specifications of an ENS, a subsystem, and a function are broken down into specifications of subsystems, functions, and transition requirements, respectively. This modular construction makes it easy to specify a large-scaled and complicated ENS piece by piece and provide a reusability of specifications. This also allows a step-wise test generation from SENS specifications for unit testing, integration testing, and system testing.

3.1.5. Lightweight Description of Time Constraints. We consider a timer as a component of a subsystem and then specify time constraints using events and actions with the timer. SENS provides a timer variable which is declared with a time-value, a time-unit, and a condition formula for counting up the timer.

3.2. Syntax of SENS. Here, we give the brief explanation of SENS description using the following example embedded system.

Example 1. Consider an illustrating system in Figure 3. There are 2 kinds of equipment, namely, classes A and B. Let us consider a simple ENS consisting of 3 subsystems A_1, B_1, and B_2, where A_1 is an instance of A and B_1 and B_2 are instances of B. A subsystem of class A sends a message m after 2 minutes from starting its operation. Each subsystem of class B turns on its lamp when receiving message m. This system can be described in the SENS specification of Pseudocode 1.

The SENS declarations consist of the following declarations (refer to Appendix A for the full syntax of SENS).

(i) Main Class Declaration. File `<main.cls>` specifies the main class. The main class represents the whole system consisting of subsystems and the environment. In the file `<main.cls>` of Pseudocode 1, the classes A and B are imported in line 1, and subsystems A_1, B_1, and B_2 are instantiated in lines 6-7. Environment variables are declared in the `Var` part. Data types, environment variables, and channels declared in the main class are referred in all class declarations. Invariants (logical expressions) for the entire system are declared in the main class. In the example `<main.cls>`, global data type `onoff_status` is declared in lines 3-4.

(ii) Class Declaration. Files `<A.cls>` and `<B.cls>` of Pseudocode 1 specify classes A and B, respectively. A class other than the main class specifies a class of subsystems. Local data types, variables, channels, and timers for the subsystem are declared in the class declaration. In `<A.cls>`, local variable op and timer `t` are declared in lines 6–9, and a function of the subsystem is declared in lines 11–14.

For data types, variables, channels, and timers externally declared and internally used, the files to declare them are imported with the `Import` statement and their names are described in the `Extern` part. The main class file is imported

```
<main.cls>
(1)  Import A, B;
(2)  Class main
(3)  Data
(4)   onoff_status {on,off};
(5)  Var
(6)    B1, B2:B;
(7)    A1:A(B1, B2);
<A.cls>
(1)  Import B;
(2)  Extern
(3)  Data
(4)    onoff_status; //from main
(5)  Class A(arg1 : B, arg2 : B)
(6)  Var
(7)    op : onoff_status {off};
(8)  Timer
(9)    t 2min;
(10)
(11)  Function SendingM
(12)  Require
(13)    ~op==off && op==on : null: t!start;
(14)    op==on : t?ring : arg1.ch!m,arg2.ch!m;
<B.cls>
(1)  Extern
(2)  Data
(3)   onoff_status; //from main
(4)  Class B
(5)  Data
(6)   m_type {m};
(7)  Var
(8)    op : onoff_status {off};
(9)    lamp : onoff_status {off};
(10)  Channel
(11)    ch : m_type;
(12)
(13)  Function Lighting
(14)  Require
(15)    op==on && lamp==off : ch?m:lamp==on;
(16)    op==off -> lamp==off;
```

Pseudocode 1: An example SENS specification.

to all class files by default. A class can have parameters used as constants in the class specification. In line 1 of `<A.cls>`, class B is imported since class B is used in the parameter declaration of A. In lines 3-4, data type `onoff_status` is described in the `Extern` part.

(iii) Variable and Data Type Declarations. A variable is declared with its name, type, initial value, and test declaration (values and bounds for testing) in the part following to `Var`. Data types are `int` and `double` with ranges, and enumerated data types are defined in the `Data` part. In the `Data` part, a type is defined as a set of values. In SENS, only finite and discrete values are used. In the example`<main.cls>`, a global data type, `onoff_status`, which has two elements `on` and `off`, is declared in lines 3-4. In `<B.cls>`, a local data type, `m_type`, which has the only element `m`, is declared

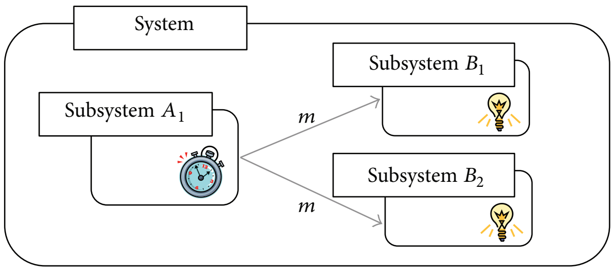


Figure 3: An example embedded system.

in lines 5-6. Using the test declaration beginning with the keyword`testedwith`, one can define the specific values of variables to be considered in a testing phase. For example, consider the following variable declaration:

```
Var v1 double [-2,5] {0}
        testedwith {-2,0..5 (0.5)};
```

variable `v1` is defined as follows: its type is double, its range is $-2 \le$ `v1` ≤ 5, its initial value 0, and its tested values are -2 and a number from 0 to 5 in increments of 0.5. In our semantics of SENS, each variable of a subsystem takes one value, that is, an element of its domain. If the declaration of a variable has a test declaration, the domain is defined by the test declaration. Otherwise, the domain is defined by the data type of the variable.

(iv) Channel and Timer Declarations. Channels are used to describe communications between subsystems. A channel is declared with its name and message type. In lines 10-11 of `<B.cls>`, a channel `ch` for receiving message `m` is declared. A timer is declared with its name, time-value, unit, and conditions (logical formula) for counting up. When the condition is omitted, the condition is always true. For example, consider the following timer declarations:

```
Timer
  t1 5 min;
  t2 5 hour [(v==dry) || (v==cool)];
```

Timer `t1` is declared to count up 5 minutes. Timer `t2` is declared to count up 5 hours, but the counting is performed only when the value of `v` is `dry` or `cool`.

The event is a message receiving (channel-name?expr) or timeout (timer-name?`ring`). Similarly, the action is a message sending (channel-name!expr), timer starting (timer-name!start), or timer reset (timer-name!clear). When specifying a communication between subsystems, a message sending (action) is described using both a recipient's object name and a channel name, while a message receiving (event) is described using only a channel name.

(v) Function and Requirement Declarations. Functions are declared in the class declaration or in individual function declaration files. In the function declaration, constants, data types, variables, and timers can be defined locally. A function contains a set of requirements.

A requirement is an expression representing an invariant or a 3-tuple of a precondition, an event, and a postcondition or/with/without actions. (More precisely, the third part of the transition requirement is a postcondition or an action or actions or a postcondition with an action or a postcondition with actions.) We call the 3-tuple requirement a transition requirement. The BNF description for a transition requirement is given in Algorithm 1 (Refer to Appendix A.4).

The precondition and postcondition are logical expressions. In the transition requirement declaration, the event part can contain no more than one event (it can also be `null` which is a symbol showing that no event occurs), while the action part can contain multiple actions. We assume that a transition requirement contains no more than one event and actions related to the same timer.

Expressions are constructed with constants, variables, and operators. Expressions have logical comparison and arithmetic operators that general programming languages such as `C` have. In addition, expressions in SENS have the logical implication operator (->) and the previous state operator (~). Especially, expression ~ `v` represents the previous value of `v`.

3.3. *Semantics of SENS.*

A SENS specification describes requirements of an ENS consisting of subsystems communicating each other. We model the system specified by SENS using a labeled transition system. Figure 4 illustrates our system modeling where time constraints in SENS are modeled using our timer models that will be shown later. First, each instance corresponding to a subsystem in the given specification is modeled. When the class corresponding to the subsystem contains timer variables, we compose the subsystem model and its timer models. Next, an entire ENS is modeled as a parallel composition of all subsystems. The transition system modeled from the SENS specification is used for generating test cases and test sequences.

In Section 3.3.1, we give the modeling of timers. In Section 3.3.2, we give the modeling of a subsystem and explain the composition of the subsystem and the timer. In Section 3.3.3, we explain the composition of subsystems.

3.3.1. *Modeling of Timers.*

A timer is specified in the timer declaration with its timer name, number (time), unit, and conditions for counting up in SENS. We consider two kinds of models M_1 and M_2 for timers expressed in Figures 5(a) and 5(b). We can use simpler timer model M_2 for state elimination, while we can use timer model M_1 for considering the relation of a time length between timers (refer to Section 3.3.2.).

Definition 2 (timer model). We model the behavior of each timer T_i as a labeled transition system $(S_{T_i}, \Sigma_{T_i}, R_{T_i})$ with $R_{T_i} \subseteq S_{T_i} \times \Sigma_{T_i} \times S_{T_i}$, where S_{T_i} denotes a set of states, Σ_{T_i} denotes a set of labels, and R_{T_i} denotes a labeled transition for T_i. $(S_{T_i}, \Sigma_{T_i}, R_{T_i})$ for M_1 and M_2 are defined as follows:

(i) $S_{T_i} = \{\texttt{ready}, 0, 1, \ldots, \texttt{max}, \texttt{over}\}$ in M_1;

(ii) $S_{T_i} = \{\texttt{ready}, \texttt{run}, \texttt{over}\}$ in M_2;

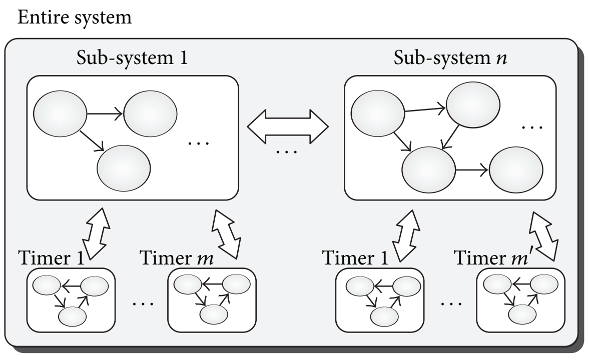
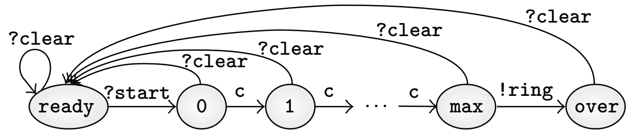


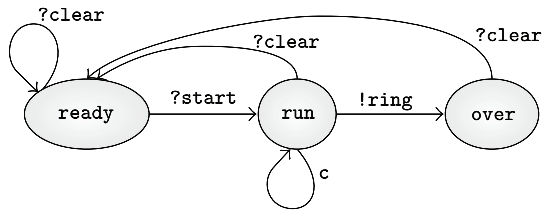
FIGURE 4: A model composition for the entire ENS.

```
requirement-declaration::=
      invariant | transition-requirement
invariant::= expr
transition-requirement::=
      pre-condition ":" event ":"
          (post-condition | (action ("," | action)*)
          | (post-condition ","(action ("," | action)*)))
```

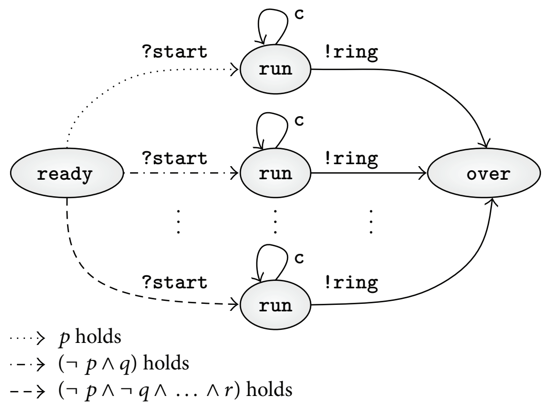
ALGORITHM 1: The BNF description for a transition requirement.



(a) Model M_1



(b) Model M_2



(c) Model M_2' for timers with multiple conditions

FIGURE 5: Timer Models.

(iii) Σ_{T_i} = {?start, ?clear, !ring, c} in M_1, M_2.

In models M_1 and M_2, states ready and over, respectively, represent states in which timer T_i is ready and counted over. In model M_1, state "h" ($0 \leq h \leq$ max) represents a state in which timer T_i has counted up h times. In model M_2, we reduce states 0,..., max to state run representing a state in which timer T_i is running, for the state elimination.

In both models, each state is changed to state ready when receiving message clear. State ready is changed to state 0 in M_1 (resp., state run in M_2) when receiving message start. State max in M_1 (resp., state run in M_2) is changed to stateover when sending message ring. We define that transitions labeled with !*start* and c occur only if the condition for the timer holds. If the condition is not described in the timer declaration, the condition is considered to be "true".

If there are multiple conditions $p, q, \ldots, r$, we define transitions as represented in Figure 5(c). In the figure, transitions labeled with ?clear are eliminated for simplicity. In the figure, the first, second, and third transitions labeled with c occur when conditions p, q, and r, respectively, hold.

3.3.2. Modeling of Subsystems. A subsystem is specified in the class declaration in SENS. For a given specification, we model the behavior of each subsystem as a labeled transition system in two steps. First, the behavior of a subsystem C_i except timers is modeled as a labeled transition system $M(C_i)$. Next, the behavior of a subsystem including timers is modeled as $M_T(C_i)$ which is a composition of $M(C_i)$ and models of the timers.

Definition 3 (subsystem model). We model the behavior of a subsystem C_i, except for its timers, as $M(C_i) = (S_{C_i}, \Sigma_{C_i}, R_{C_i})$ with $R_{C_i} \subseteq S_{C_i} \times \Sigma_{C_i} \times S_{C_i}$. Given a SENS specification, the set of states S_{C_i}, the set of labels Σ_{C_i}, and the set of transitions R_{C_i} are obtained by Definitions 15, 16, and 17 of Appendix B.1.

Example 4. Consider the example specification of subsystem C_A in Pseudocode 1. Figure 6 illustrates the subsystem model for C_A which satisfies the specification. Given SENS specification for class A, the states are (˜op=off, op=off), (˜op=off, op=on), (˜op=on,op=on), and (˜op=on, op=off) from variable op and previous state variable ˜op used in the specification by Definition 15. By Definition 16, the labels are t!start, t?ring, B[1]. ch!m, and B[2]. ch!m with the events and actions used in the specification. Each transition in $M(C_A)$ satisfies the two conditions in Definition 17. For example, $s \rightarrow s'$ with s = (˜op=off,op=off) and s' = (˜op=on,op=off) cannot be a transition of $M(C_A)$, since it does not hold the condition for previous state variables; that is, $s \models$ (op=on), if and only if $s' \models$ (˜op=on) by Definition 17.

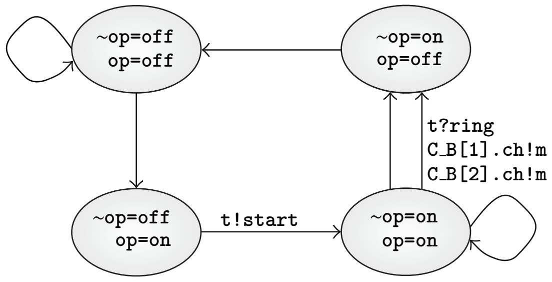


FIGURE 6: An example model for system C_A.

We model the behavior of a subsystem C_i with its timers $T_1, T_2, \ldots, T_m$ as an interleaving composition of the subsystem model and timer models as follows:

$$M_T(C_i) = M(C_i) \parallel M(T_1) \parallel M(T_2) \parallel \cdots \parallel M(T_m), \quad (2)$$

where $M(C_i)$ denote the subsystem model explained in Definition 3 and $M(T_j)$ denotes the model for timer T_j $(1 \le j \le m)$ explained in Section 3.3.1. We inductively define the above model composition, that is,

(1) $M(C_i) \parallel M(T_j)$ in Definition 18,

(2) $M(CT_i) \parallel M(T_k)$ in Definition 19, where $M(CT_i) = M(C_i) \parallel M(T_1) \parallel \cdots \parallel M(T_j)$ with $j < k$.

And the composition rule is defined in Definitions 18 and 19 of Appendix B.2.

Example 5. Consider again the example specification of subsystem C_A in Pseudocode 1. Figure 7 shows a part of the composed model of $M(C_A)$ in Figure 6 and timer model M_2 for timer t. The transitions labeled with t_`start` and t_`ring` correspond to rules (a1) and (a2) in Definition 18, that is, message synchronization. The transition labeled with `c` corresponds to rule (a4) in Definition 18.

3.3.3. System Model. Now we explain the model for the entire system. Assume that the entire system consists of k subsystems. We define the model for the system as the parallel synchronized composition of labeled transition systems $M_T(C_i)$ representing all subsystems C_i $(1 \le i \le k)$. The composition of two subsystems is defined in Definition 20 and the entire system model $M_T(C_1) \parallel \cdots \parallel M_T(C_n)$ is obtained by an inductive composition. (When a transition of a subsystem has multiple message sending, we assume that its composition is synchronously performed in our modeling.) The composition rule is defined in the definition of Appendix B.3.

Example 6. Consider the example specifications of subsystems C_A and $C_B[1]$ in Pseudocode 1. Figure 8 shows an example composition of transitions in $M_T(C_A)$ and $M_T(C_B[1])$.

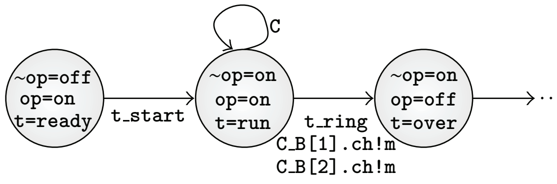


FIGURE 7: An example model for system C_A with a timer.

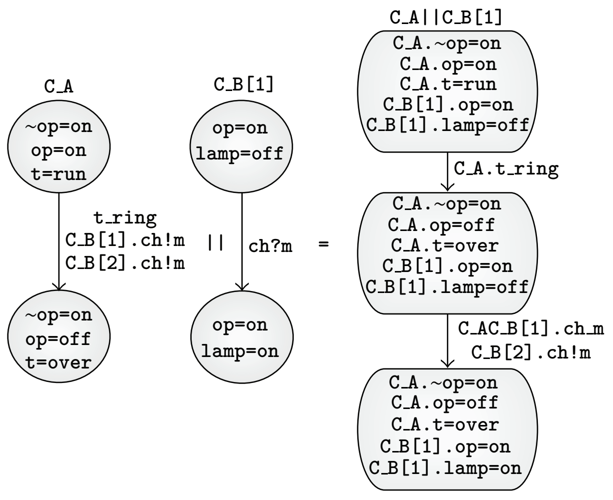


FIGURE 8: An example transition composition for C_A and $C_{B[1]}$.

4. Automatic Test Generation from the SENS Specification

In this section, we present the methods for a given SENS specification to automatically generate test cases and test sequences in Sections 4.1 and 4.2, respectively.

4.1. Test Case Generation

4.1.1. Concept. In our approach, we find all test cases by using a SAT solver. A SAT solver is a tool to solve the SAT (satisfiability) problem which determines whether there is a value assignment to satisfy a given Boolean formula in CNF (conjunctive normal form). Although this problem is an NP-complete problem, there have been very high-speed SAT solvers (e.g., MiniSAT [5]) for practical sized problems. Using a SAT solver, we can find distinct and enormous test cases quickly that satisfy SENS specifications.

4.1.2. Test Cases. Given a SENS specification that describes a target system, every transition of the system model satisfies all requirements in the given specification, as mentioned in the previous section. Therefore, we can consider that each transition t of the system model corresponds to a one-step test case. That is, the prestate (resp., events) of t is a valid input state (resp., system inputs) of a test case and the poststate (resp., action) of t is a valid output state (resp., system output) of the test case. We formally define a test case as follows.

Definition 7 (test case). Consider a given specification $\mathcal{S}$. A test case is a tuple,

$$(input, event, action, output), \quad (3)$$

corresponding to a transition of the system model M such that $M \models \mathcal{S}$. In other words, a test case is an assignment that makes every transition requirement of $\mathcal{S}$ hold.

Note that, if the given specification is for a subsystem module, subsystems, and an entire ENS, derived test cases are, respectively, used for unit testing, integration testing, and system testing.

4.1.3. Translation of SENS to CNF. For generating test cases, we first translate all the requirements in the given SENS specification to a CNF formula which becomes an input of a SAT solver. Since the SENS specification is a set of logical constraint formulas, we can translate it to CNF according to the semantics of SENS.

There are 3 main steps of the translation procedure as follows. First, we introduce fresh propositional variables to construct the state space of the system model. Second, transition requirements and invariants in the SENS specification are translated into logical formulas. Third, we collect all of the constraints presented in logical formulas. The constraints are connected with conjunction and the result is converted to CNF. (Every propositional logic formula can be converted into an equivalent formula, that is, in CNF. However, by naive algorithms where the transformation is based on rules of De Morgan's laws and the distributive law, the size of CNF can be exponentially increased. To get a compact CNF formula, we used the *Tseitin-transformation* [6] after translating SENS to a propositional logic formula.) Due to space limitation, the details of all translation rules are given in Appendix C.

4.1.4. Generating Test Cases. Once we obtain the CNF formula from the given SENS specification, we apply a SAT solver to the CNF formula to find an assignment. As mentioned above, the assignment that makes requirements of the SENS specification true corresponds to a test case.

Algorithm 2 describes our method for generating test cases. We use a SAT solver iteratively to find all possible assignments and convert them to test cases that cover all possible transitions of the system model.

4.1.5. Filtering. The proposed method can generate all possible test cases with 100% coverage for requirements of a given specification. On the other hand, the number of complete test cases can be very huge. In order to extract efficient test cases, we also give several filtering mechanisms in our method. In the current version of our tool, a user can select test cases satisfying a precondition of transition requirements, test cases containing a communication label, and test cases whose timer label is`ring`, which means the test cases related to a time constraint.

Example 8. Consider again the example specification of subsystem C_A in Pseudocode 1. Each transition of the model for system C_A in Figure 7 corresponds to a test case. Especially, ((`~op=off,op=on`), ∅, {`t!start`}, (`~op=on,op=on`)) correspond to a test case satisfying a precondition of the transition requirement `<~op==off&&op==on:null:t!start>` in line 13 of the specification. On the other hand, ((`~op=on,op=on`),{`t?ring`}, {`C_B[1].ch!m,C_B[2].ch!m`}, (`~op=on,op=off`)) correspond to a test case containing timer label ring and communication labels.

As shown in Table 3, the number of all test cases for C_A is 176. Among them, test cases satisfying a precondition are 10, which is less than 6% of the entire test cases. Moreover, among them, 2 test cases can be extracted, respectively, with a communication label and timer label `ring`.

4.2. Test Sequence Generation

4.2.1. Concept. As mentioned in Section 4.1, our test generator can generate all possible test cases for a given SENS specification. Since the number of complete test cases can be very huge, it is hard to run tests by inputting all the test cases separately. Therefore, it is practical to extract test sequences and focus on specific variables interested for testing. To deal with this problem, we propose a test sequence generator for appropriate testing goals given by users.

4.2.2. Test Sequences and Testing Goals. In our method, we generate test sequences from two inputs: given a SENS specification and a testing goal. First, we define a test sequence as follows.

Definition 9 (test sequence). A test sequence is a sequence of test cases, whose adjacent output and input are the same. This means that a test sequence is a sequence,

$$\begin{gathered}\langle (input_1, event_1, action_1, output_1) \\ \ldots, (input_n, event_n, action_n, output_n)\rangle,\end{gathered} \quad (4)$$

where $output_i = input_{i+1}$, for $i = 1, \ldots, n-1$.

Next, a "testing goal" is a key feature that helps to generate appropriate test sequences. This also allows us to reduce the size of space and time for searching test sequences. In the current version of our method, a user can specify the following 3 features as a testing goal: "key variables" for a testing purpose, a list of "milestone" states forming a backbone of the sequence, and a "maximum length" between two adjacent milestones. We formally define a testing goal as follows.

Definition 10 (testing goals). Given a SENS specification, a testing goal $G = \langle V, S, Ls\rangle$, where V is a set of key variables, S is a list of milestone states, and Ls is a list of maximum lengths such that

(1) *a* "key variable" is a variable that we focus on testing in the given specification;

(2) *a* "milestone" is a state that we want each test sequence to pass through;

```
     Data: A SENS specification
     Result: A Set of test cases
(1)  begin
(2)      Translate the given SENS specification to a logical formula $f$ in conjunctive normal form (CNF)
         based on translation rules in Appendix C;
(3)      Input $f$ to a SAT solver;
(4)      if $f$ is satisfiable then
(5)          The SAT solver outputs a solution $m$ (value assignment) corresponding to one test case;
(6)          Add the solution $m$ to a set of solutions;
(7)          Let $f \leftarrow f \wedge \neg m$ and return Step 2 to search for another solution;
(8)      else
(9)          The SAT solver declares $f$ is unsatisfiable;
(10)         Then there are no more solutions;
(11)         Translate previous solutions to test cases and stop;
(12)     return The set of test cases;
```

Algorithm 2: Our algorithm of test case generation.

(3) *a* "maximum length," max of a run from a milestone $s \in S$ to the next milestone $s' \in S$, denoted by $s \xrightarrow{\leq \max} s'$, is the maximum number of transitions needed for the run.

We say that a test sequence $\langle (i_1, e_1, a_1, o_1), \ldots, (i_n, e_n, a_n, o_n) \rangle$ satisfies a testing goal $G = \langle V, S = \langle s_1, \ldots, s_k \rangle, Ls \rangle$, if and only if (1) the first input state i_1 and the last output state o_n of the test sequence, respectively, are equal to the first milestone s_1 and the last milestone s_k, (2) all other milestone states exist in the input states of the test sequence in order, and (3) the length between the milestone states in the test sequence is less than the maximum length defined in Ls. In addition, value assignments only for the key variables appear in the generated test sequences satisfying the testing goal.

Example 11. If the list of milestones is $\langle s_1, s_2, s_3 \rangle$, then we can specify the maximum length in total as the $\xrightarrow{\leq \max_1} \cdot \xrightarrow{\leq \max_2}$, where $\max_1$ is the maximum of transitions which can be taken to go through from the milestone s_1 to the milestone s_2, and $\max_2$ is the maximum of transitions which can be taken to go through from the milestone s_2 to the milestone s_3.

Let us consider the simple example specification of system B in Pseudocode 1. Assume a testing goal whose key variables $V = \{\texttt{op}, \texttt{lamp}\}$, milestones $S = \langle s_1 = (\texttt{op=off}, \texttt{lamp=off}), s_2 = (\texttt{op=on}, \texttt{lamp=on}) \rangle$, and $s_1 \xrightarrow{\leq 2} s_2$. Then,

$$\begin{gathered}\langle ((\texttt{op=off}, \texttt{lamp=off}), \emptyset, \emptyset, (\texttt{op=on}, \texttt{lamp=off})), \\ ((\texttt{op=on}, \texttt{lamp=off}), \texttt{ch?m}, \emptyset, (\texttt{op=on}, \texttt{lamp=on})) \rangle\end{gathered}$$

is a test sequence satisfying the testing goal.

4.2.3. Generating Test Sequences. Our approach for test sequence generation has three main steps. First, from a given SENS specification, we first generate all test cases by our method in Algorithm 2. Then, the obtained test cases correspond to all transitions of the target system model.

Second, we synthesize all the transitions into a transition system (graph). Each state of this transition system is a tuple of variable assignments. Each edge is a transition relation from a prestate to a poststate, and a label of each edge is an event that makes the transition occur. The transition system of the specification is stored in a file, while the testing goal is stored in a separate file.

Third, given the transition system and the testing goal in two files, we compute all possible desired test sequences that satisfy the testing goal; that is, we search the set of sequences starting from a given milestone state to another milestone state whose distance is less than the given maximum length by tree searching algorithms including exhaustive BFS (breath first search) and DFS (depth first search) algorithms. Each resulting test sequence is treated as a test scenario. In our test sequence generation, we reduce the searching state space of the system model by an equivalence partitioning based on the key variables of the given testing goal and search the sequences containing milestone states.

In our TGSENS, test sequences can be generated from both the original transition system and the filtered test cases. In the test sequence generation from the original transition system, the obtained test sequences obviously keep the transition relation of the original system model. In the test sequence generation from the filtered test cases, test sequences including transitions eliminated by filtering are not generated. However, since the filtering is defined by formal rules, the effect by filtering is clear for users who recognize that test sequences with filtering options will be obtained. For this reason, we believe that it does not affect the main results of our research, in other words, generations of reliable test cases/sequences.

5. Implementation and Experiments

5.1. Implementation. We implemented our approach of SENS-based automatic test generation in the tool called TGSENS. The tool TGSENS consists of the following four parts as illustrated in Figure 2. (1) The first part translates the specification written in SENS into a CNF formula. (2) The second part searches and collects all assignments that satisfy the CNF formula by using a SAT solver as its core engine. (3) The third part translates the obtained assignments into test cases which are expressed with variables used in the original SENS specification. (4) The forth part searches the test sequences from the output file of the third part and given testing goals. The resulting test sequences can be viewed by two ways: a textual format or a graphical figure using a dot file viewer.

Currently, for the function level (corresponding to the function declaration in SENS) and the subsystem level (corresponding to the class declaration in SENS), we have completed the implementation of the first part in TGSENS but not yet for the system level. That means that the implementation for the composition of subsystems for the entire system level is our future work. Hence, the current version of our tool can perform the test generation for unit testing.

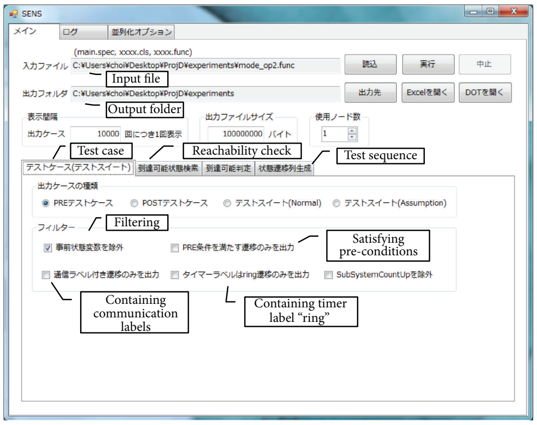
Our tool was implemented on Windows XP for both ×86 and ×64 environments. The first, third, and forth parts were implemented with a Java language. The second part was implemented based on MiniSAT [5] with C++. The size of the implemented programs is totally about 20,000 lines, respectively, 16, 000, 3, 000, 500, and 340 lines for each part.

We also implemented the user interface (in Japanese) of TGSENS given in Figure 9. Given an input SENS specification, our tool has execution options to generate output files of exhaustive test cases, filtered test cases, and test sequences under a given testing goal. TGSENS also has a function to check a reachability of states and generate path sequences among the input states.

5.2. Experiments: Specification in SENS. We applied SENS to specify system requirements of real air-conditioning systems by Company D. In general, there are specifications with different levels. In Company D, specifications are divided into "instruction manuals," "functional specifications," and "detailed specifications," and each specification is broken into functions. For example, as to specifications of certain indoor equipment, an instruction manual consists of about 10 functions with 30 pages, and a functional specification and a detailed specification, respectively, consist of about 80 functions with 400 pages.

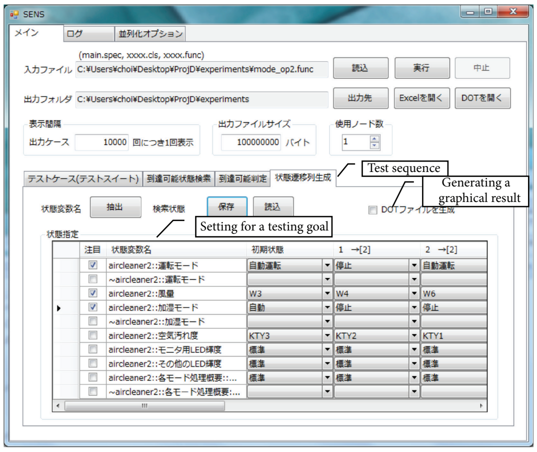
We translated several specifications in a natural language with tables and figures to SENS specifications. The target specifications were for two modules F_h and F_i in an instruction manual and one function F_j in a detailed specification of an indoor equipment, one function F_l in a functional specification of an air-cleaning system, and three functions F_m, F_n, and F_o in a functional specification of a controller.

For all specifications, this translation needed 1 or 2 days by one person who is one of our SENS developers. Almost all (70–80%) of the translation time was for understanding the real specifications and confirming whether the SENS specifications are valid with industrial engineers, while the rest of the time (several hours) was for the actual description. In addition, we held a one-day seminar for industrial engineers to explain how to describe the SENS specifications and we



(main.spec, xxxx.cls, xxxx.func)
入力ファイル C:¥Users¥choi¥Desktop¥ProjD¥experiments¥mode_op2.func
出力フォルダ C:¥Users¥choi¥Desktop¥ProjD¥experiments
表示間隔 出力ケース 10000 回につき1回表示
出力ファイルサイズ 100000000 バイト
使用ノード数 1
出力ケースの種類
[✓ PREテストケース] POSTテストケース テストスイート(Normal) テストスイート(Assumption)
フィルター
[✓ 事前状態変数を除外] PRE条件を満たす遷移のみを出力
通信ラベル付き遷移のみを出力 タイマーラベルはring遷移のみを出力 SubSystemCountUpを除外

(a) The main interface for generating test cases.



(main.spec, xxxx.cls, xxxx.func)
入力ファイル C:¥Users¥choi¥Desktop¥ProjD¥experiments¥mode_op2.func
出力フォルダ C:¥Users¥choi¥Desktop¥ProjD¥experiments
表示間隔 出力ケース 10000 回につき1回表示
出力ファイルサイズ 100000000 バイト
使用ノード数 1
状態指定

注目	状態変数名	初期状態	1 →[2]	2 →[2]
✓	aircleaner2::運転モード	自動運転	停止	自動運転
	~aircleaner2::運転モード			
✓	aircleaner2::風量	W3	W4	W6
✓	aircleaner2::加湿モード	自動	停止	停止
	~aircleaner2::加湿モード			
	aircleaner2::空気汚れ度	KTY3	KTY2	KTY1
	aircleaner2::モニタ用LED輝度	標準	標準	標準
	aircleaner2::その他のLED輝度	標準	標準	標準
	aircleaner2::各モード処理概要::...	標準	標準	標準
	~aircleaner2::各モード処理概要:...			

(b) The interface for generating test sequences.

FIGURE 9: The GUI interface of TGSENS.

confirmed that after the one-day seminar, the engineers were easily able to use SENS to specify embedded systems by themselves.

Example 12. We show a part of an example SENS specification for function F_l in Pseudocode 2. F_l is a mode change function for an air-cleaning system by Company D. The original specification of this function was written in Japanese with A_4 sized 4 pages and includes 5 tables and 4 transition diagrams. We transformed this document into about 100 lines of a SENS specification. The SENS specification consists of 8 variables (including 6 global variables, 1 local variable, and 1 channel), 20 invariants, and 8 transition specifications. Lines 16–18 specify invariants and lines 20–22 specify transition specifications. All 8 variables have discrete values and the details of variables and their ranges are given in Table 1.

Table 2 shows the scale of the SENS specifications for function F_h-F_o: the number of lines except comments and blank lines, the number of declared variables and channels, and the number of requirements including invariants and transition requirements. As to the specification of F_o, the size of requirements was relatively long since variables of the array

```
<AC.cls>
Class AC
Data
  Type_opMode      {Off, Auto, Turbo, Pollen, Calm, Normal};
  Type_humidMode   {Off, Auto, Cont, High};
  Type_airVolume   {W0, W1, W2, W3, W4, W5, W6};
  Type_airPurity   {KTY1, KTY2, KTY3, KTY4, KTY5};
  Type_Brightness  {Normal, Dark, Off};
  Type_switch      {On};
Var
  Mode_A: Type_opMode; //Mode of operation
  Mode_B: Type_humidMode; //Mode of humidity
  AV : Type_airVolume; //Actual air volume
  AP : Type_airPurity; //Air purity
  BM : Type_Brightness; //Brightness for monitor LED
  BO : Type_Brightness; //Brightness for other LED
Channel
  C : Type_switch; //Indicating pushing a button
  ···

<Mode_Change.func>
Bundled AC;
Extern
Data
  Type_opMode, Type_humidMode,
  Type_airVolume, Type_airPurity,
  Type_Brightness, Type_switch;
 Var
  Mode_A, Mode_B, AV, AP, BM, BO;
 Channel
  C;

 Function Mode_Change
 Var
   BU : Type_Brightness; //User-defined brightness
 Require
  Mode_A==Auto -> (AV==W1 || AV==W2 || AV==W3 || AV==W4 || AV==W5);
  Mode_A==Auto && Mode_B==Off && AP==KTY1 -> AV==W1;
  Mode_A==Calm -> AV==W1;
  ···
  Mode_A!=Off && BU==Normal: C?On : BU==Dark;
  Mode_A!=Off && BU==Dark: C?On : BU==Off;
  true : C?On : Mode_A==~Mode_A;
  ···
```

Pseudocode 2: An example SENS specification for an air-cleaning system

type with a big size are used. In the future, our SENS description could be more compact by preparing for-statement or while-statement descriptions. For comparison, we also show the number of pages of the original specifications in a natural language in Table 2.

Although we, here, showed the experimental results only for air-conditioning systems, we also have other case studies applying SENS to specify embedded systems such as an electric pot and an automatic dispenser. For instance, we applied SENS to describe a requirement specification for an electric pot (http://www.sessame.jp/workinggroup/WorkingGroup2/POT_Specification.htm), which has been openly used as an example embedded system by SESSAME (http://www.sessame.jp) (Society of Embedded Software Skill Acquisition for Managers and Engineers) of Japan. The original requirement specification was written in Japanese including state transition diagrams with 14 pages and consisting of 14 functions. We described approximately 1,100 lines of the SENS specifications for all the functions, which include 28 variables, 5 timers, and 7 channels in the class specification and totally 386 requirements for all function specifications. Through our case studies, we confirmed that SENS is efficiently applicable to not only the air-conditioning systems but also other general embedded systems.

In addition, by describing the specifications in SENS, we found several undefined and under-specified requirements

Table 1: Variables in the example specification for function F_l.

Variable	Variable description	Range
Mode_A	Mode of operation	6
Mode_B	Mode of humidity	4
AV	Actual air volume	7
AP	Air purity	5
BM	Brightness for monitor LED	3
BO	Brightness for other LED	3
BU	User-defined brightness	3
C	Channel for indicating pushing a button	3

in the original specifications for the target air-conditioning system. Through the feasibility study, we could confirm both the description ability and facility of SENS. Engineers in Company D plan to use our tool in real development processes. We anticipate that using SENS increases the quality of specifications by preventing the undefinition, the ambiguity, and the inconsistency in the specifications.

5.3. Experiments: Test Generation. In this section, we describe several case studies of generating test cases and test sequences from given example specifications <A.cls> and <B.cls> in Pseudocode 1 and F_i, F_j, F_k, and F_l, using our tool TGSENS. F_k is a specification whose variable range is reduced from that of F_j for testing. The experiment was performed in the following environments: Intel Core 2 Duo P8600 2.4 GHz, 3.0 GB Memory, Microsoft Windows XP SP3, and Sun Java Runtime Environment 1.6.0_24.

Table 3 describes the result of applying TGSENS to test case generation. In the table, we show the size of SENS specifications, the size of CNF, the number of test cases, and the execution times to generate test cases for each part of TGSENS. For the size of a SENS specification, we represent the number of states and the number of transition labels in the table. Here, the number of states indicates the one of the state space constructed by all possible variable assignments in the specification. The number of transition labels indicates the number of labels in the system model constructed from the specification.

Example 13. Consider the case of F_l. The number of possible states (including states not satisfying the specification) for F_l is $6 \times 4 \times 7 \times 5 \times 3^3 = 68{,}040$ from Table 1. The number of transition labels is 2 as shown in Table 3. Then the number of all possible state transitions (those that do not satisfy the specification are included) is $68{,}040 \times 2 \times 68{,}040 \approx 9.3 \times 10^9$. From such a large number of transitions, TGSENS figured out 77,700 transitions satisfying the given specification as test cases within 12 seconds. For instance, as for the filtering option of including a communication label, TGSENS selected 8,100 test cases (≈10.5% of all the test cases) within 1 second.

For F_j (resp., F_k), TGSENS took only about 26 (resp., 1) minutes to generate approximately 900,000 (resp., 160,000) test cases. For example, the specification of F_j has 13 variables including 3 previous state variables and those value ranges are 3, 5, 5, 4, 5, 5, 2, 3^5, 3^5, 3, 3, 3, and 3. Then, the number of states for F_j is $2 \times 3^{15} \times 4 \times 5^4 = 71{,}744{,}535{,}000 \approx 7.2 \times 10^{10}$. On the other hand, the numbers of states for F_k and F_l are 45,000 and 68,040, respectively. F_j has a feature of including variables whose range are large, like 3^5, and thus the number of states is excessively large compared to those of other specifications. The execution result of F_j indicates that the specification has $71{,}744{,}535{,}000 \approx 7.2 \times 10^{10}$ states and 30 transition labels. Then, the number of all possible state transitions for F_j is $71{,}744{,}535{,}000 \times 30 \times 71{,}744{,}535{,}000 \approx 1.5 \times 10^{23}$. Since the number of state transitions for F_j is much larger than those for F_k and F_l, the CNF size for F_j is much larger than those for F_k and F_l. Accordingly, F_j needed much execution time to search test cases. Although, with such a huge amount of state space, it is very impossible to manually check all cases by human, TGSENS can correctly do that and takes only about 26 minutes.

We next describe the result of applying TGSENS to test sequence generation. Table 4 shows the results using the SENS specification for function F_l with 3 example testing goals, G1–G3. In the table, for each testing goal, we show the number of milestone states, the maximum lengths between milestones, the number of key variables of the given testing goal, the number of test sequences, and the execution time to generate test sequences.

Example 14. Pseudocode 3 shows the input file to specify the testing goal G3 in TGSENS. We can also set the testing goal directly in the GUI interface of TGSENS shown in Figure 9(b). The testing goal G3 specifies the following:

(i) key variables (declared in lines 25–28): three variables Mode_A, Mode_B, and AV in the specifi cation.

(ii) The list of milestones (declared in lines 4–19): $\langle s_1, s_2, s_3 \rangle$, where s_1 is a state in which Mode_A=Auto, s_2 is a state in which Mode_A=Off, and s_3 is a state in which Mode_A=Auto.

(iii) maximum lengths between milestones (declared in lines 22-23): $s_1 \xrightarrow{\leq 2} s_2$ and $s_2 \xrightarrow{\leq 2} s_3$.

When the specification for F_l and the testing goal G3 are given, a transition system of this function is first obtained from the result of test case generation. Then, TGSENS automatically searches desired test sequences satisfying the testing goal from the transition system. Pseudocode 4 shows a part of the generated test sequences.

The resulting transition system for F_l has 1,800 states and 77,700 transitions. From the transition system, TGSENS figured out 28 (resp., 32) sequences from s_1 to s_2 (resp., s_2 to s_3) and thus totally 896 (=28 × 32) test sequences satisfying the testing goal only in 3 minutes.

6. Related Work

Model-based testing technique has been an attractive research area in software engineering field recently [7–9]. Model-based testing is a technique that derives a desired test case from a model or a specification of a system. As modeling

Table 2: Scales of example SENS specifications.

System	Function	SENS Spec. Number of lines	SENS Spec. Number of variables	SENS Spec. Number of req.	Original spec. Number of pages
Indoor equip.	F_h	12	5	7	1
Indoor equip.	F_i	22	4	6	1
Indoor equip.	F_j	71	6	22	2
Air cleaner	F_l	101	7	28	4
Controller	F_m	77	6	22	3
Controller	F_n	52	6	4	4
Controller	F_o	428	10	47	4

Table 3: Experiment results for test case generation.

System function	SENS Spec.		CNF		Test cases	Execution times (sec.)		
	Number of states	Number of trans. labels	Number of literals	Number of clauses	Number of test cases	SENS to CNF	Search all assignments	Translate to test cases
A	12	17	25	68	176	0.044	0.031	0.056
B	4	3	11	19	19	0.050	0.006	0.013
F_i	12	36	36	141	1,330	0.069	0.053	0.125
F_j	$\approx 7.2 \times 10^{10}$	30	620	62,845	893,155	0.175	1565.90	56.57
F_k	4.5×10^4	30	102	406	157,874	0.078	62.24	21.77
F_l	$\approx 6.8 \times 10^4$	3	91	715	77,700	0.151	9.609	1.929

```
[AllTransitionFileName]
 mode_op2.csv;

 //Milestone List
  [MilestoneList]
 //first Milestone
AC::Mode_A#Auto,
AC::AV#W3,
AC::AP#KTY3,
AC::Mode_B#Auto,
AC::Mode_Change::BU# Normal,
AC::BM#Normal,
AC::BO#Normal;

//second Milestone
AC::Mode_A#Off;

//third Milestone
AC::Mode_A#Auto;

//maximum lengths between two milestones
[StepNumList]
2,2;

[SelectVarNameList]
AC::Mode_A,
AC::AV,
AC::Mode_B;
```

Pseudocode 3: An example testing goal.

Table 4: Experiment results for test sequence generation.

	(1)	(2)	(3)	Number of sequences	Exe. time
G1	1	2	$\cdot \xrightarrow{\leq 2} \cdot$	25	1.3 sec.
G2	1	2	$\cdot \xrightarrow{\leq 3} \cdot$	156	8 min.
G3	3	3	$\cdot \xrightarrow{\leq 2} \cdot \xrightarrow{\leq 2} \cdot$	896	3 min.

(1) Number of key variables, (2) number of milestones, and (3) Maximum lengths.

makes it possible to refine or check functionality of target systems, we can expect to obtain correct test cases generated from valid models in earlier phases of development. Thus model-based test generation has been attracting the attention of model-driven software development approaches. Model-based testing has two technical issues: modeling target systems appropriately and generating test cases effectively.

As for the modeling, many techniques have been proposed, even if we focus on formal modeling only. Models are described based on contracts, abstract data types, process algebras, labeled transition systems, data flow, and so on. These concepts are implemented using specification languages in tools like VDM [7], Z [10], B [11], Spec# [12], and JML [13]. With most modeling languages, a user has to specify the systems' behaviors individually.

Test case generations from models have been proposed in several ways. Here, test cases include arguments to execute specific functions, pairs of possible inputs and expected

```
[Trace: 0]
[AC::Mode_B#Auto, AC::Mode_A#Auto, AC::AV#W3]
[AC::Mode_B#Cont, AC::Mode_A#Auto, AC::AV#W1]
[AC::Mode_B#Off, AC::Mode_A#Off, AC::AV#W0]
[AC::Mode_B#Off, AC::Mode_A#Auto, AC::AV#W1]
[AC::Mode_B#High, AC::Mode_A#Auto, AC::AV#W1]

[Trace: 1]
[AC::Mode_B#Auto, AC::Mode_A#Auto, AC::AV#W3]
[AC::Mode_B#Cont, AC::Mode_A#Auto, AC::AV#W1]
[AC::Mode_B#Off, AC::Mode_A#Off, AC::AV#W0]
[AC::Mode_B#Off, AC::Mode_A#Auto, AC::AV#W3]
[AC::Mode_B#High, AC::Mode_A#Auto, AC::AV#W1]

[Trace: 2]
[AC::Mode_B#Auto, AC::Mode_A#Auto, AC::AV#W3]
[AC::Mode_B#Cont, AC::Mode_A#Auto, AC::AV#W1]
[AC::Mode_B#Off, AC::Mode_A#Off, AC::AV#W0]
[AC::Mode_B#Off, AC::Mode_A#Auto, AC::AV#W2]
[AC::Mode_B#High, AC::Mode_A#Auto, AC::AV#W1]
...
```

Pseudocode 4: An example output: Test sequences.

outputs, or event sequences. Test cases are searched by tracing executions of models, selecting random values with filtering, or generating candidates with some algorithms.

Here, we describe previous works on model-based testing. Spec Explorer [9] generates test cases by state exploration based on multiple strategies from contract-like specifications Spec# on a Microsoft. NET framework. In TGV [8] by INRIA, as a part of the CADP (Construction and Analysis of Distributed Processes) package, test graphs are generated from labeled transition systems. Uppaal-Tron [14] generates test sequences for real-time systems using the model checker UPPAAL, and an industrial case study of Uppaal-Tron targeted an electronic refrigerator controller. T-VEC [15] is a coverage-based test generation tool of Simulink/MathWorks which has been used to identify a fault in the Mars Polar Lander.

Some model-based test generators use constraints solving to generate test cases. In [16], a test generator based on autofocus is proposed. Autofocus [17] is a graphical tool for model-based development of distributed systems. By translating a specification into a constraint logic program, this generator can deal with all the possible execution traces of the model. TestEra [18] is a test generation framework for Java programs. By considering pre-/postconditions for executing methods as constraints, TestEra generates test cases with SAT-solvers. Exhaustive generation is available through certain options.

Most previous studies targeted the testing of programs or protocols. Although TGV has case studies for small-sized embedded applications, there are only a few studies on the large-scale embedded network systems targeted in our proposed project. Different from other works, we have proposed the original specification language, SENS, focusing on property-based specifications and the tool TGSENS to automatically generate test cases and sequences focusing on testing goals for embedded network systems.

7. Conclusion

In this paper, we have proposed a method to automatically generate test cases and sequences based on the specification of embedded networked systems (ENSs). We first proposed a formal specification language SENS whose features are (1) property-based, (2) test-oriented, (3) object-oriented, (4) modular description, and (5) lightweight description of time constraints. By the "property-based" feature such that requirements are property-based constraints holding in the entire ENS, logical properties for the system specification are clear and modifications and reuse of the specifications are easy since a part of specifications less affect the entire specifications. Thus, SENS has an advantage on the efficient description and reuse of specifications for large and complicated ENSs. The "object-oriented" and "modular" feature enables SENS to structurally describe requirement specifications of ENSs consisting of a number of embedded subsystems/equipment. The "lightweight" feature on description of time constraints leads to readily specify embedded systems sensitive to time requirements. In addition, the "test-oriented" feature is helpful for automatic test generation from SENS specifications. To the best of our knowledge, there has been no other formal specification language having all the above features for ENSs. Using SENS, we can efficiently describe the requirements of ENSs.

We next proposed the methods for automatic test generation in the tool TGSENS whose features are summarized as follows: (1) TGSENS takes specifications of an ENS written in SENS as its input and automatically generates test cases and test sequences; (2) the test case generation problem is translated into a satisfiability problem, and test cases are generated by iteratively using a SAT solver; (3) the test sequences are also automatically obtained based on a testing goal, that is, key variables, milestone states, and a desired maximum length of test sequences, given by a user. We finally applied TGSENS to generate tests for the real air-conditioning systems, and our tool rapidly succeeded in generating about 900,000 test cases in 30 minutes and extract 28 test sequences in 3 minutes.

We expect that using TGSENS can improve the quality of test cases and thus enhance the quality of ENSs. Previously in industrial fields, test cases have often been designed in a manual way. Such the test design depending on testing engineers' skills seldom covers all the requirements of the complicated ENSs. On the other hand, TGSENS can automatically and mechanically generate test cases from given formal SENS specifications. Both the description of formal specifications and the automatic test generation lead to improving the quality of test cases. Actually, we found several undefined and under-specified requirements by describing the SENS specifications for real industrial embedded systems as noted in Section 5.2. In addition, TGSENS can generate test cases with 100% coverage for requirements of the given SENS specification by an exhaustive search using a SAT-based solver. TGSENS also supports feasible test generation by user-specified test filtering and test sequence generation.

The successive work of our project is the improvement of our test generator in order to deal with the composition of subsystems. One of the most important aspects of test generation based on formal specifications and models for industrial systems is "scalability." To overcome this problem, we plan to tackle a higher-speed and more scalable test generation using parallel computing. We also plan to find a constructive way of generating test suites of the whole system from test suites of subsystems as a future work. We also plan to apply the model checking technique to generate more target-specific test cases and sequences.

Appendices

A. Syntax of SENS

A.1. Main Class Declaration. See Pseudocode 5.

A.2. Class Declaration. See Pseudocode 6.

A.3. Variable, Channel, and Timer Declarations. See Pseudocode 7.

A.4. Function and Requirement Declarations. See Pseudocode 8.

A.5. Expression. See Pseudocode 9.

B. Semantics of SENS

B.1. Modeling of Subsystems

States For $M(C_i)$. Let $V_{C_i} = \{v_1, \ldots, v_n\}$ denote a set of all variables used in the SENS specification for the subsystem C_i. That is, V_{C_i} is a set of all variables declared in the variable declaration of the class declaration for C_i, and the extern declaration of the class-file-declaration for C_i. For each variable v_i $(1 \le i \le n)$, let $d(v_i)$ denote a set of all values of v_i declared in the specification. When considering a test model, let $d(v_i)$ be a set of all values in the test declaration for v_i.

In addition, we also define a set of variables with previous state operator "~." Let $V'_{C_i} = \{v'_1, \ldots, v'_{n'}\}$ $(\subseteq V_{C_i})$ denote a set of every variable $v'_i \in V_{C_i}$ whose previous state variable $\sim v'_i$ is used in the specification.

Definition 15 (states of $M(C_i)$). S_{C_i} is a set of all states s such that

(i) s is an $(n + n')$-tuple of value assignments for all variables in V_{C_i} and V'_{C_i}; that is, $s = (v_1 = a_1, \ldots, v_n = a_n, \sim v'_1 = a'_1, \ldots, \sim v'_{n'} = a'_{n'})$, $a_j \in d(v_j)$ $(1 \le j \le n)$, $a'_j \in d(v'_j)$ $(1 \le j \le n')$;

(ii) s satisfies all conditions specified as invariants in the requirement declaration of the class declaration for C_i and the invariant declaration.

Hereafter we denote $s \models (v_i = a_i)$ if the value for v_i is assigned to a_i at state s. Otherwise $s \not\models (v_i = a_i)$ and thus $s \models (v_i \neq a_i)$ since each variable is assigned to one value at the same time.

Labels For $M(C_i)$. Let $L(C_i)$ be a set of all events and actions declared in the class declaration and the class-file-declaration for subsystem C_i. There are three kinds of labels: input labels corresponding events, output labels corresponding actions, and label τ.

(i) Input labels are *channel-name?expr* and *timer-name*`?ring` and represent message receiving.

(ii) Output labels are *channel-name!expr*, *timer-name*`!start`, and *timer-name*`!clear` and represent message sending.

(iii) Label τ means an internal transition with no message receiving and sending.

Definition 16 (labels of $M(C_i)$). Σ_{C_i} is a set of labels such that

$$\Sigma_{C_i} = \{L \mid L \in \mathscr{P}(L(C_i)), |\text{in}(L)| \le 1\}, \qquad \text{(B.1)}$$

where in(L) denotes a set of input labels in L.

Each L $(\in \Sigma_{C_i})$ is a set of labels in a transition, and we call L a transition label. Note that each transition label can contain at most one input label and multiple output labels by the syntax of SENS. Transition label $\emptyset$ in Σ_{C_i} corresponds to τ.

Transitions For $M(C_i)$. Consider a transition requirement $\langle P : e : Q, A\rangle$ where P is precondition, e is event, Q is

```
main-class-file-declaration:: = import-declaration main-class-declaration
import-declaration:: = “Import” (‘”file-name‘” “,”)+ “;”
main-class-declaration:: = “Class main”
          (constant-declaration | data-declaration| env-instance-declaration
          | channel-declaration | timer-declaration)+
          (invariant-declaration)?
env-instance-declaration:: = “Var” (env-variable-declaration | instance-declaration)+
env-variable-declaration:: =
     env-variable-name “:” type-name (“{” value “}”)?  (test-declaration)? “;”
     | env-variable-name “[” number “..” number “]” “:” type-name (“{“value (“,” value)* “}”)? (test-declaration)? “;”
     | env-variable-name “:” (“int” | “double”)
            (“[“min-value “,” max-value “]”)? (“{”value“}”)? (test-declaration)? “;”
     | env-variable-name “[“number “..” number“]” “:” (“int” | “double”)
             (“[”min-value “,” max-value “]”)? (“{”value (“,” value)* “}”)? (test-declaration)? “;”
instance-declaration::=
     object-name “:” class-name (“(” parameter-value (“,” parameter-value)* “)”)? “;”
     | object-name “[” number “..” number “]” “:” class-name
      (“{” “(”value (“,” value)* “)” (“,” “(” value (“,” value)* “)”)* “}”)? “;”
invariant-declaration::= “Inv” (expr “;”)+
```

Pseudocode 5: BNF for main class declaration.

```
class-file-declaration::=  class-declaration
          (import-declaration | extern-declaration)?
class-declaration::= “Class” name (parameter-declaration)?
         (constant-declaration | data-declaration | variable-declaration
         | channel-declaration  | timer-declaration)+
         (function-declaration | invariant-declaration)+
extern-declaration::= “Extern”
         (“Const” const-name (“,” const-name)* “;”
         | “Data” type-name (“,” type-name)* “;”
         | “Var” variable-name (“,” variable-name)* “;”
         | “Timer” timer-name (“,” timer-name)* “;”
         | “Channel” channel-name (“,” channel-name)* “;”)+
parameter-declaration:: = “(” parameter-name “:” type-name (“,” parameter-name (“:” type-name)* “)”
constant-declaration::= “Const” (const-name expr “;”)+
```

Pseudocode 6: BNF for class declaration.

```
data-declaration::= “Data” (type-name “{” value (“,”value)* “};”
variable-declaration::=“Var”
     (variable-name “:” type-name (“{” value “}”)? (test-declaration)? “;”
     | variable-name “[” number “..” number “]” “:” type-name  (“{” value (“,” value)* “}”)? (test-declaration)? “;”
     | variable-name “:” (“int” | “double”)
        (“[” min-value “,” max-value “]”)? (“{” value “}”)?  (test-declaration)? “;”
     | variable-name “[” number “..” number “]” “:” (“int” | “double”)
         (“[” min-value “,” max-value “]”) ? (“{”value (“,” value)*“}”)?  (test-declaration)? “;”
     )+
test-declaration::=  “testedwith” “{” (number | number “..” number (“(” number “)”)?)
           (“,” (number | number “..” number (“(“ number “)”)?) )* “}”
channel-declaration::=  “Channel” (channel-name “:” type-name “;”
          | (channel-name “[” number “..” number “]” “:” type-name “;”) +
timer-declaration:: =
        “Timer” (timer-name number unit “[” predicate “] “;”
               |timer-name (number unit “[” predicate “],”)*number unit “[” predicate “]”) “;”)+
unit::= “msec” | “sec” | “min” | “hour” | “day”
```

Pseudocode 7: BNF for variable, channel, and timer declarations.

```
function-file-declaration::= “Bundled” class-name extern-declaration function-declaration
function-declaration::=
    “Function” function-name
        (constant-declaration | data-declaration | variable-declaration | timer-declaration)*
        “Require” (requirement-declaration “;”)+
requirement-declaration::= invariant | transition-requirement
invariant::= expr
transition-requirement::= pre-condition “:” event “:”
        (post-condition | (action (“,” | action)*) | (post-condition “,” (action (“,” | action)*)))
precondition::= expr
postcondition::= expr
event::= channel-name “?” expr | timer-name “?” “over” | “null”
action::= channel-name “!” expr  | timer-name “!” “clear”  | timer-name “!” “start”
```

Pseudocode 8: BNF for function and requirement declerations.

```
expr::= expr “– >” expr | expr “||” expr | expr “&&” expr | expr “ == ” expr | expr expr “ != ” expr
        | expr “<” expr | expr “<=” expr | expr “>” expr | expr “>=” expr | expr “+” expr | expr “–” expr
        | expr “∗” expr | expr “/” expr | expr “%” expr | “+” expr | “–” expr | “!” expr
        | number | const | variable | “~” variable | “true” | “false” | “ (”expr“)”
```

Pseudocode 9: BNF for expression declaration.

postcondition, and A is actions. We say that a transition $s \xrightarrow{l} s'$ satisfies a transition requirement $\langle P : e : Q, A\rangle$, if and only if the following holds:

$$(s \models P) \wedge (e \in l) \longrightarrow \left(s' \models Q\right) \wedge (A \subseteq l); \qquad \text{(B.2)}$$

that is, if state s satisfies the precondition P and event e is an input label of l, then next state s' should satisfy the postcondition Q and actions A should be output labels of l. Hereafter $s \xrightarrow{l} s'$ denotes a labeled transition (s, l, s').

Definition 17 (transitions of $M(C_i)$). R_{C_i} is a set of all $s \xrightarrow{l} s' \in S_{c_i} \times \Sigma_{c_i} \times S_{c_i}$ that satisfies the following two conditions (we say that such a transition satisfies the given specification):

(i) for each state variable $\sim v_i \in s'$, $s' \models (\sim v_i = a_i)$, if and only if $s \models (v_i = a_i)$;

(ii) $s \xrightarrow{l} s'$ satisfies all the transition requirements of the requirement declaration in the class declaration for subsystem C_i.

B.2. Modeling of Subsystems with Timers

Definition 18 (composition of a subsystem and a timer). $M(C_i) \parallel M(T_j)$ is defined as follows. Hereafter, let $mg \in \{clear, start, ring\}$ and let p_j denote the condition for timer T_j:

(i) states: $(s, t) \in S_{C_i} \times S_{T_j}$;

(ii) labels: $L \subseteq (L(C_i) - \Sigma_{T_j} \cup \{c, T_j\_mg\})$, $|\text{in}(L)| \leq 1$;

(iii) transitions: for each $s \xrightarrow{l_S} s' \in R_{C_i}$ and $t \xrightarrow{l_T} t' \in R_{T_j}$, the following transitions exist:

(a1) $(s, t) \xrightarrow{l_S - \{T_j?mg\} \cup \{T_j\_mg\}} (s', t')$,
if $l_S \ni T_j?mg$ and $l_T = !mg$;

(a2) $(s, t) \xrightarrow{l_S - \{T_j!mg\} \cup \{T_j\_mg\}} (s', t')$,
if $l_S \ni T_j!m$ and $l_T = ?mg$;

(a3) $(s, t) \xrightarrow{l_S} (s', t)$,
if $l_S \cap \{T_j?mg, T_j!mg\} = \emptyset$ and $l_T = c$;

(a4) $(s, t) \xrightarrow{c} (s, t')$,
if $l_S \cap \{T_j?mg, T_j!mg\} = \emptyset$, $l_T = c$, and $s \models p_j$; that is, the condition for timer T_j holds at state s.

The transitions of rules (a1) and (a2) represent the composition of message synchronization. The transitions of rules (a3) and (a4) represent the interleaving composition of transitions of subsystem C_i and timer T_j, respectively. By this interleaving composition with a subsystem and a timer, the composed model can cover all combinations of time length in the model.

Definition 19 (composition of a subsystem and timers). $M(CT_i) \parallel M(T_k)$ with $M(CT_i) = M(C_i) \parallel M(T_1) \parallel \cdots \parallel M(T_j)$ is defined as follows. Note that $mg \in \{clear, start, ring\}$ and p_k denotes the condition for timer T_k:

(i) states: $(s, t) \in S_{CT_i} \times S_{T_k}$;

(ii) labels: $L \subseteq (\Sigma_{C_i} - \Sigma_{T_k} \cup \{c, T_k\_mg\})$, $|\text{in}(L)| \leq 1$;

(iii) transitions: for each $s \xrightarrow{l_S} s' \in R_{C_i}$ and $t \xrightarrow{l_T} t' \in R_{T_k}$, the following transitions exist:

(b1) $(s,t) \xrightarrow{l_S-\{T_k?mg\}\cup\{T_k\_mg\}} (s',t')$,
if $l_S \ni T_k?mg$ and $l_T = !mg$;

(b2) $(s,t) \xrightarrow{l_S-\{T_k!mg\}\cup\{T_k\_mg\}} (s',t')$,
if $l_S \ni T_k!mg$ and $l_T = ?mg$;

(b3) $(s,t) \xrightarrow{l_S} (s',t)$,
if $l_S \cap \{T_k?mg, T_k!mg, c\} = \emptyset$ and $l_T = c$;

(b4) $(s,t) \xrightarrow{c} (s,t')$,
if $l_S \cap \{T_k?mg, T_k!mg, c\} = \emptyset$, $l_T = c$, $s \vDash p_k$, and $s \nvDash p_1 \vee \cdots \vee p_j$;

(b5) $(s,t) \xrightarrow{c} (s',t')$,
if $l_S = c$, $l_T = c$, and $s \vDash p_k$.

The transitions of rules (b1) and (b2) represent the composition of message synchronization. The transitions of rules (b3) and (b4) represent the interleaving composition of transitions of subsystem CT_i and timer T_k, respectively. Transition of rule (b5) represents the synchronization of counting up timers. Therefore, the length relation of timers can be modeled by using timer model M_1 in Section 3.3.1.

B.3. Composition of Subsystems

Definition 20 (composition of subsystem models). $M_T(C_i) \parallel M_T(C_j)$ for subsystems C_i and C_j is defined as follows:

(i) states: $(s, t) \in S_{CT_i} \times S_{CT_j}$, where $S_{CT_i}(S_{CT_j})$ is the set of states of $M_T(C_i)(M_T(C_j))$;

(ii) labels: for each $L_i(\in \Sigma_{CT_i})$ and $L_j(\in \Sigma_{CT_j})$,

$$L \subseteq L_i \cup L_j - \{ch?mg, ch!mg, c\} \cup \{ch\_mg\}, \qquad \text{(B.3)}$$

where $ch?mg$ and $ch!mg$ are, respectively, input and output labels representing receiving and sending massage mg with communication channel ch between system C_i and C_j. Here, ch_mg denotes a label after composing input and output labels with the same name.

(iii) transitions: let $L_C = \{ch?mg, ch!mg | ch$ be a communication channel between C_i and $C_j\}$ and $L_N = \{ch?mg, ch!mg | ch$ is not a communication channel between C_i and $C_j\}$. in(L) and out(L), respectively, denote a set of input labels and output labels in L. The following transitions exist for each $s \xrightarrow{l_i} s' \in R_{CT_i}$ and $u \xrightarrow{l_j} u' \in R_{CT_j}$:

(c1) $(s,u) \xrightarrow{\tau} (s',u')$, if $l_i, l_j \subset \{\tau, c\}$;

(c2) $(s,u) \xrightarrow{l_i} (s',u)$, if $l_i \subseteq L_N$;

(c3) $(s,u) \xrightarrow{l_j} (s,u')$, if $l_j \subseteq L_N$;

(c4) $(s,u) \xrightarrow{l} (s',u)$, if $l = \text{in}(l_i) \not\subset L_C$;

(c5) $(s,u) \xrightarrow{l} (s,u')$, if $l = \text{in}(l_j) \not\subset L_C$;

(c6) $(s,u') \xrightarrow{l} (s',u')$, if

(i) $\text{in}(l_i) = \{ch?mg\}$, $\text{out}(l_j) \ni ch!mg$,
(ii) $\text{out}(l_i) \cap L_C = \emptyset$,
(iii) $(\text{out}(l_j) - ch!mg) \cap L_C = \emptyset$,
(iv) $l = \text{out}(l_i) \cup \text{out}(l_j) \cup \{ch\_mg\} - \{ch!mg\}$;

(c7) $(s',u) \xrightarrow{l} (s',u')$, if

(i) $\text{in}(l_j) = \{ch?mg\}$, $\text{out}(l_i) \ni ch!mg$,
(ii) $\text{out}(l_j) \cap L_C = \emptyset$,
(iii) $(\text{out}(l_i) - ch!mg) \cap L_C = \emptyset$,
(iv) $l = \text{out}(l_i) \cup \text{out}(l_j) \cup \{ch\_mg\} - \{ch!mg\}$.

The transition of rule (c1) represents the composition of internal transitions labeled with τ and c. The transitions of rules (c2) and (c3) represent the composition of internal transitions labeled with external communication channels. The transitions of rules (c4)–(c7) are for the composition of two transitions including message synchronization. In the composition, we interpret the meaning of a transition with input and output labels of subsystems as illustrated in Figure 10. In the figure, transition $(s,u) \xrightarrow{l2'} (s,u')$ corresponds to rule (c4) where $l2' = \text{in}(l2)$. Transition $(s,u') \xrightarrow{m,l3} (s',u')$ corresponds to rule (c6) where $l3 = l1 \cup l2$.

C. Translating SENS to Propositional Logic

Here, we explain how to translate given a SENS specification to a propositional logic formula.

C.1. Variable Declaration.

For each variable `v` and its value `t` in the SENS specification, we introduce two auxiliary propositional variables representing the value of `v` in the current state as `t` (denoted by `v#t`) and the value ofv in the next state as `t` (denoted by `@v#t`). In addition, if the variable `v` appears in an expression in the form `~v`, then we also introduce two auxiliary propositional variables such as `~v#t` and `@~v#t`. The value `t` of `v` is an element in the domain of `v`; the domain is defined by test declaration or the type of the variable.

All variables of the SENS specification have the following two constraints: (1) a variable must take a value in its domain at any time and (2) a variable cannot take two values at the same time. Moreover, if a variable with ~ operator appears in an expression, the variable has the following constraint: the value of a variable with ~ operator in the next state equals the value of the variable without ~operator in the current state.

In this translation, we need to represent these constraints as logical formulas by using auxiliary propositional variables. For example, consider variable op of `<A.cls>` in Pseudocode 1. The constraints to variable op are as follows:

(i) $(\texttt{op\#on} \vee \texttt{op\#off}) \wedge (\texttt{@op\#on} \vee \texttt{@op\#off})$;

(ii) $(\neg\texttt{op\#on} \vee \neg\texttt{op\#off}) \wedge (\neg\texttt{@op\#on} \vee \neg\texttt{@op\#off})$;

(iii) $(\texttt{@\~{}op\#on} \Leftrightarrow \texttt{op\#on}) \wedge (\texttt{@\~{}op\#off} \Leftrightarrow \texttt{op\#off})$.

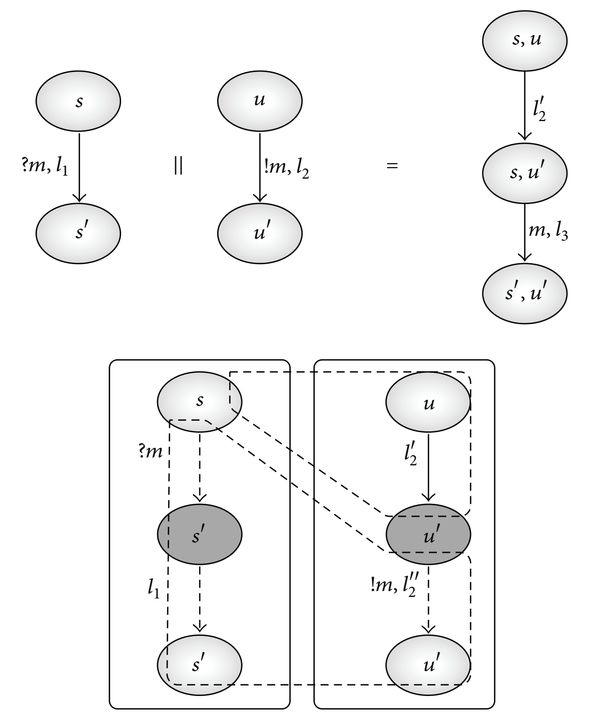


Figure 10: A transition composition of subsystems.

C.2. Timer Declaration. We translate the timer declaration step-by-step considering the three parts: a subsystem side, a timer side, and a composition of the subsystem and the timers.

First, on the subsystem side, we prepare four kinds of propositional variables as follows:

(i) `SubCountUp`: a counting up event occurring in a subsystem is prepared; there exists only one variable of this kind for each subsystem;

(ii) `t?ring`: a subsystem receives a counting over event from its timer `t`;

(iii) `t!clear`: a subsystem does an action by sending a reset message to its timer `t`;

(iv) `t!start`: a subsystem does an action by sending a startup message to its timer `t`.

We assume that no more than one event or action related to the same timer `t` occurs at the same time. We represent this constraint as follows:

(i) $(\neg\texttt{t?ring} \vee \neg\texttt{t!clear}) \wedge (\neg\texttt{t?ring} \vee \neg\texttt{t!start})$
$\wedge(\neg\texttt{t!clear} \vee \neg\texttt{t!start})$
$\wedge(\neg\texttt{t?ring} \vee \neg\texttt{SubCountUp})$
$\wedge(\neg\texttt{t!clear} \vee \neg\texttt{SubCountUp})$
$\wedge(\neg\texttt{t!start} \vee \neg\texttt{SubCountUp})$.

Next, on the timer side, we introduce auxiliary propositional variables for two events, an action, a counting-up event, and the states of each timer. The propositional variables prepared for timer `t` are as follows:

(i) `t?clear/t?start`: for reset/start-up event from the subsystem,

(ii) `t!ring`: an action of sending the counting over message,

(iii) `t?CountUp`: a counting-up event of the timer,

(iv) `t#ready/t#run/t#over`: the current timer state,

(v) `@t#ready/@t#run/@t#over`: the next timer state.

Timer `t` in the subsystem has the following constraints and we represent them using prepared propositional variables as follows:

(i) the events and action of timer `t` do not occur at the same time:

(a) $(\neg\texttt{t?clear} \vee \neg\texttt{t?start})$
$\wedge(\neg\texttt{t?clear} \vee \neg\texttt{t!ring})$
$\wedge(\neg\texttt{t?start} \vee \neg\texttt{t!ring})$
$\wedge(\neg\texttt{t?clear} \vee \neg\texttt{t?CountUp})$
$\wedge(\neg\texttt{t?start} \vee \neg\texttt{t?CountUp})$
$\wedge(\neg\texttt{t!ring} \vee \neg\texttt{t?CountUp})$;

(ii) at any time, the state of a timer is one of three states, *ready*, *run*, and *over*:

(a) $(\texttt{t\#ready} \vee \texttt{t\#run} \vee \texttt{t\#over})$
$\wedge(\texttt{@t\#ready} \vee \texttt{@t\#run} \vee \texttt{@t\#over})$,

(b) $(\neg\texttt{t\#ready} \vee \neg\texttt{t\#run})$
$\wedge(\neg\texttt{t\#ready} \vee \neg\texttt{t\#over})$
$\wedge(\neg\texttt{t\#run} \vee \neg\texttt{t\#over})$
$\wedge(\neg\texttt{@t\#ready} \vee \neg\texttt{@t\#run})$
$\wedge(\neg\texttt{@t\#ready} \vee \neg\texttt{@t\#over})$
$\wedge(\neg\texttt{@t\#run} \vee \neg\texttt{@t\#over})$;

(iii) the transition relation of the timer models is the same as the one in Figure 5:

(a) $\texttt{t?clear} \rightarrow \texttt{@t\#ready}$,
(b) $\texttt{t?start} \Leftrightarrow (\texttt{t\#ready} \wedge \texttt{@t\#run})$,
(c) $\texttt{t!ring} \Leftrightarrow (\texttt{t\#run} \wedge \texttt{@t\#over})$,
(d) $\texttt{t?CountUp} \rightarrow (\texttt{t\#run} \wedge \texttt{@t\#run})$,
(e) $(\texttt{t\#ready} \wedge \neg\texttt{t?clear} \wedge \neg\texttt{t?start} \rightarrow \texttt{@t\#ready})$
$\wedge(\texttt{t\#run} \wedge \neg\texttt{t?clear} \wedge \neg\texttt{t!ring}$
$\wedge\neg\texttt{t?CountUp} \rightarrow \texttt{@t\#run})$
$\wedge(\texttt{@t\#over} \wedge \neg\texttt{t?clear} \rightarrow \texttt{@t\#over})$;

(iv) when the timer declaration has a condition for counting up, that is, a logical formula `cond`, the following two formulas are added:

(a) $\texttt{t?start} \rightarrow \texttt{cond}$,
(b) $\texttt{t?CountUp} \rightarrow \texttt{cond}$.

Finally, we describe constraints of composition between a subsystem and timers.

(i) Events and actions of a subsystem are associated with actions and events of each timer `t`:

(a) `t!clear` $\Leftrightarrow$ `t?clear`;
(b) `t!start` $\Leftrightarrow$ `t?start`;
(c) `t?ring` $\Leftrightarrow$ `t!ring`.

(ii) For timers of each subsystem, all the timers satisfying their counting up conditions count up at the same time:

(a) $(\texttt{t1?CountUp} \rightarrow \texttt{SubCountUp}) \wedge(\texttt{t2?CountUp} \rightarrow \texttt{SubCountUp}) \wedge\cdots\wedge(\texttt{tn?CountUp} \rightarrow \texttt{SubCountUp})$;
(b) $\texttt{SubCountUp} \rightarrow (\texttt{t1?CountUp} \vee \texttt{t2?CountUp} \vee\cdots\vee \texttt{tn?CountUp})$;
(c) $\texttt{SubCountUp} \rightarrow ((\texttt{cond}_{\texttt{t1}}) \rightarrow \texttt{t1?CountUp}) \wedge(\texttt{cond}_{\texttt{t2}}) \rightarrow \texttt{t2?CountUp}) \wedge\cdots\wedge(\texttt{cond}_{\texttt{tn}}) \rightarrow \texttt{tn?CountUp})$, where $\texttt{cond}_{\texttt{ti}}$ is the counting up condition of timer `ti`.

(iii) The counting up of timers in a subsystem and a change of the values of variables in the subsystem do not occur at the same time. Consider variable `op` of `<A.cls>` in Pseudocode 1. This constraint is represented by the following formula:

(a) $(\texttt{SubCountUp} \wedge \texttt{op\# on} \rightarrow \texttt{@op\# on}) \wedge(\texttt{SubCountUp} \wedge \texttt{op\# off} \rightarrow \texttt{@op\# off})$.

C.3. Channel Declaration. For events through a channel, we prepare propositional variables. A variable denoted by `c?m` shows that the event of receiving message `m` through channel `c` occurs. In addition, for each subsystem, prepare one propositional variable representing that no event occurs (denoted by `null#true`).

No more than one event occurs at the same time in a subsystem where we consider the counting over of a timer and the counting up of a subsystem as events. We need to represent this constraint of event as logical formulas by using prepared variables. For example, consider the subsystem `<A.cls>` in Pseudocode 1. The subsystem has two events, a timer-over event and a counting up the timer. So, the constraint of events in `<A.cls>` is translated as follows:

(i) $\texttt{t?ring} \vee \texttt{SubCountUp} \vee \texttt{null\#true}$;
(ii) $(\neg\texttt{t?ring} \vee \neg\texttt{SubCountUp}) \wedge(\neg\texttt{t?ring} \vee \neg\texttt{null\#true}) \wedge(\neg\texttt{SubCountUp} \vee \neg\texttt{null\#true})$.

For actions as well as events, we prepare propositional variables. A variable denoted by `obj.c!m` shows that a subsystem takes the action of sending message `m` to a district subsystem `obj` througha receiver's channel `c`.

There is a constraint that a subsystem cannot take any action, if its counting-up event occurs. For example, consider the subsystem `<A.cls>`. Since it has two actions, the constraint of actions is expressed by the following formula:

(i) $(\neg\texttt{arg1.ch!m}\vee\neg\texttt{SubCountUp}) \wedge(\neg\texttt{arg2.ch!m}\vee\neg\texttt{SubCountUp})$.

C.4. Requirements. Basically, in the translation of requirements of a SENS specification, we only have to replace the requirements with logical formulas using the propositional variables prepared above. A requirement represented as a logical expression in SENS should be satisfied both in the current state and in the next state. Therefore, we translate the requirement into two logical formulas: one for the current state and the other for the next state.

The transition requirement of SENS specifications, "P: e: (Q, a)," represents a logical implication: "$(P\wedge e) \rightarrow (Q\wedge a)$." The postcondition Q is a constraint for the next state, so we translate the condition into a logical formula using propositional variables for the next state.

For example, consider the two requirements of `<B.cls>` in Pseudocode 1. They are translated as follows:

(i) $(\texttt{op\#on}\wedge\texttt{lamp\#off}\wedge\texttt{ch?m}) \rightarrow (\texttt{@lamp\#on})$;
(ii) $\texttt{op\#off} \rightarrow \texttt{lamp\#off}, \texttt{@op\#off} \rightarrow \texttt{@lamp\#off}$.

All the variables of SENS including variables of numerical types take only a finite number of values. We start explaining the translation using the following examples,
`Var a: int [0,2]; b: int [0,2];`

(i) comparison between a variable of SENS and a numerical value:

(a) the equation `a==0` is denoted by a propositional variable `a#0`, and the inequation a!=0 is denoted by a literal $\neg$`a#0`;
(b) the inequality `a<2` is translated to a logical formula $\texttt{a\#0} \vee \texttt{a\#1}$.

(ii) comparison between variables:

(a) the equation `a==b` is translated to a formula $(\texttt{a\#0} \wedge \texttt{b\#0})\vee(\texttt{a\#1} \wedge \texttt{b\#1}) \vee (\texttt{a\#2} \wedge \texttt{b\#2})$;
(b) the inequation `a!=b` is translated to a formula $\neg((\texttt{a\#0} \wedge \texttt{b\#0}) \vee (\texttt{a\#1} \wedge \texttt{b\#1}) \vee (\texttt{a\#2}\wedge\texttt{b\#2}))$;
(c) the inequality `a<b` is translated to a formula $(\texttt{a\#0} \wedge (\texttt{b\#1} \vee \texttt{b\#2})) \vee (\texttt{a\#1} \wedge \texttt{b\#2})$.

For general numerical comparison, we translate the requirement inductively. In particular, let $\exp_1 \sim \exp_2$ be a general numerical comparison in SENS, where $\sim\in \{\leq, \geq, =, \neq\}$, and $\exp_1, \exp_2$ are expressions. We transform this requirement to logical formula as follows. First, we introduce two auxiliary variables v_1 and v_2; these variables represent $\exp_1$ and $\exp_2$, respectively. The requirement $\exp_1 \sim \exp_2$ is reduced to $v_1 \sim v_2$. Second, based on the domain of input variables, we can compute the domain (lower and upper values) of v_1 and v_2. Assume that $v_1 \in [l_1, u_1]$ and $v_2 \in [l_2, u_2]$. The requirement $v_1 \sim v_2$ is translated to logical formula by a similar way as in the example above.

Conflict of Interests

The authors declare that they have no conflict of interests.

Acknowledgments

The authors thank Takao Sonoda, Shouichi Hasuike, Munekazu Huruichi, Keiichi Shimatani, and Hayato Shikata of DaiKin Industries for their great contribution in their joint project. This paper was partly supported by the JSPS-VAST bilateral research project.

References

[1] I. Craggs, M. Sardis, and T. Heuillard, "AGEDIS case studies: model-based testing in industry," in *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pp. 106–117, Nuremburg, Germany, 2003.

[2] A. Gargantini and C. L. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Software Engineering*, O. Nierstrasz and M. Lemoine, Eds., vol. 1687 of *Lecture Notes in Computer Science*, pp. 146–162, Springer, Berlin, Germany, 1999, Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering.

[3] G. Hamon, L. De Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM '04)*, pp. 261–270, Beijing, China, September 2004.

[4] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data flow testing as model checking," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pp. 232–242, Portland, Ore, USA, May 2003.

[5] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds., vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518, Springer, Berlin, Germany, 2004.

[6] G. Tseitin, "On the complexity of derivation in propositional calculus," in *Structures in Constructive Mathematics and Mathematical Logic*, A. O. Silenko, Ed., part 2, pp. 115–125, 1968.

[7] C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall International Series in Computer Science, Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1990.

[8] C. Jard and T. Jéron, "TGV: theory, principles and algorithms: a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 297–315, 2005.

[9] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with spec explorer," in *Formal Methods and Testing*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949 of *Lecture Notes in Computer Science*, pp. 39–76, Springer, Berlin, Germany, 2008.

[10] B. Potter, D. Till, and J. Sinclair, *An Introduction to Formal Specification and Z*, Prentice Hall, New York, NY, USA, 1996.

[11] A. Hoare, P. Chapron, and J. R. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, New York, NY, USA, 1996.

[12] M. Barnett, K. Rustan, M. Leino, and W. Schulte, "The Spec# programming system: an overview," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362 of *Lecture Notes in Computer Science*, pp. 49–69, Springer, Berlin, Germany, 2004.

[13] L. Burdy, Y. Cheon, D. R. Cok et al., "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, 2005.

[14] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study," in *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05)*, pp. 299–306, Jersey City, NJ, USA, September 2005.

[15] M. R. Blackburn and R. D. Busser, "T-VEC: a tool for developing critical systems," in *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS '96)*, pp. 237–249, Gaithersburg, Md, USA, June 1996.

[16] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl, "Model-based test case generation for smart cards," *Electronic Notes in Theoretical Computer Science*, vol. 80, pp. 170–184, 2003.

[17] F. Huber, B. Schätz, and G. Einert, "Consistent graphical specification of distributed systems," in *Industrial Applications and Strengthened Foundations of Formal Methods*, J. Fitzgerald, C. B. Jones, and P. Lucas, Eds., vol. 1313 of *Lecture Notes in Computer Science*, pp. 122–141, Springer, Berlin, Germany, 1997.

[18] S. Khurshid and D. Marinov, "TestEra: specification-based testing of java programs using SAT," *Automated Software Engineering*, vol. 11, no. 4, pp. 403–434, 2004.