# Automatic Verifying Approach for Product Specification using FTA

Tetsuji FUKAYA, Masayuki HIRAYAMA, Yukihiro MIHARA
Research & Development Center, Systems & Software Engineering Laboratory,
Toshiba Corporation
70 Yanagi-cho, Kawasaki, 210 JAPAN

## Abstract

*We propose a verification method for software specification. In order to avoid software faults, our method derives safety assertions using FTA, computes the behavioral graph of specification and analyzes statically whether this graph satisfies safety assertions. Moreover, when there exists an assertion which can not hold, our method localizes software design faults.*

## 1 Introduction

In order to develop high quality software, it is important to build up quality in the *specification design stage.* As the role of software in safety critical system increases, *safety* becomes an important factor among several characteristics of software quality. Many kinds of techniques have been developed to assure safety of software system. Verification is one of these techniques. However, these techniques have several limitations [1]:

• *it is difficult to define safety assertions from initial product requirements,*

• *formal verification is usually limited to the source code, and thus cannot address problems arising.*

In this paper, in order to solve these limitations, we propose a software design method suitable for verification and a formal verification method for a *software specification.* We discuss how to derive safety assertions that must be enforced in a specification and how to verify a specification with safety assertions automatically.

Moreover, we show an example of our approach applied to a practical *microwave oven* development.

## 2 Software Design Steps

### 2.1 Definition of software Primitive Objects

I. Extracts Software Primitive Objects. (SPO)

SPOs are essential functional elements of a product. In other words, these are abstract representations of items, such as "control devices", "control data", that are closely related to a target product.

II. Define *modes* and *methods* of each SPO.

*Modes* are states where SPO can change.

III. Define a specification of each SPO's *activities*.

*Methods* are operations for SPO. There are two kinds of *Methods*; **Event** and **Action. Events** are used to detect *mode* of SPO. **Actions** are used to change *mode* of SPO.

Fig. 1 shows an example of the definition of Primitive Object "Timer".

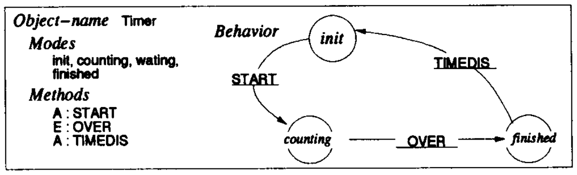


Figure 1: Definition of Primitive Object "Timer"

### 2.2 Description of a product specification

A product behavioral specification expresses a product's activities. Using State Transition Diagram, designers describe product behavioral specification.

A State Transition Diagram consists of states and transitions. Each transition has events and actions. Events and actions are described as follows:

<u>[*a Primitive Object Name, Method*]</u> ex. [Timer, START]

Fig. 2 is an example of a product behavioral specification of *microwave oven.*

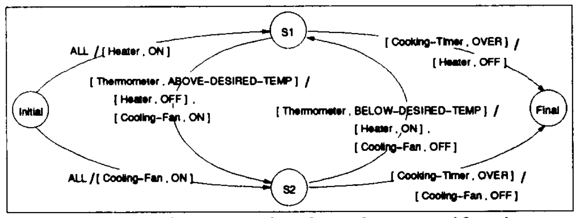

S1
ALL / [ Heater , ON ]
[ Cooking-Timer , OVER ] /
[ Heater , OFF ]
[ Thermometer , ABOVE-DESIRED-TEMP ] /
[ Heater , OFF ] ,
[ Cooling-Fan , ON ]
Initial
[ Thermometer , BELOW-DESIRED-TEMP ] /
[ Heater , ON ] ,
[ Cooling-Fan , OFF ]
Final
ALL / [ Cooling-Fan , ON ]
S2
[ Cooking-Timer , OVER ] /
[ Cooling-Fan , OFF ]

Figure 2: An example of product specification

## 3 Verification method

This method detects logical errors during the design phase of software production.

This method takes the following steps:

**I.** Generates safety assertions that must hold in a product specification,

**II.** Verifies whether a product specification satisfies defined safety assertions automatically.

### 3.1 Generation of assertions

In order to generate safety assertions, we propose HSFTA(Hardware & Software FTA). It can be applied to the system that includes hardware and software. It can generate assertions to assure fault avoidance.

• Using HSFTA, analyzes fault factors of a product including software.

• In order to avoid software fault factors, derives safety assertions which should hold in a specification. And translates these assertions into logical constraints between software functions.

#### 3.1.1 Hardware & Software FTA

The purpose of applying HSFTA(Hardware & Software FTA) to a product is to derive *safety assertions* in addition to general requirements.

HSFTA deploys top-level fault factors into software and hardware factors. It has following two stages:

1. Deploys fault factors to Primitive Object level (with conventional FTA),
2. Deploys fault factors to *modes* and *Methods* of Primitive Object (with HSFTA).

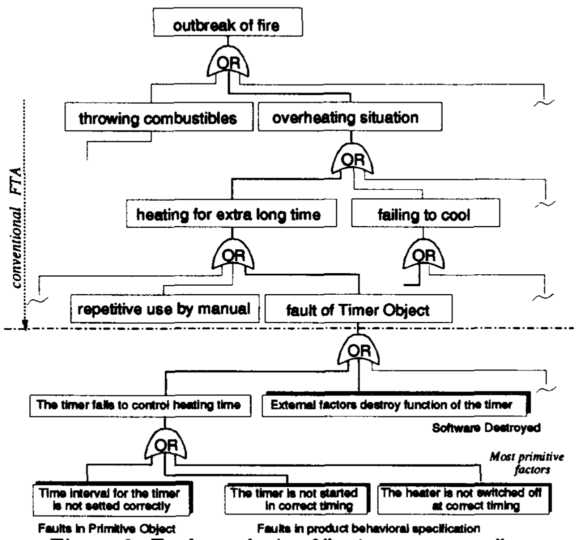

outbreak of fire
OR
throwing combustibles
overheating situation
OR
heating for extra long time
failing to cool
OR
OR
repetitive use by manual
fault of Timer Object
conventional FTA
OR
The timer fails to control heating time
External factors destroy function of the timer
Software Destroyed
OR
Most primitive factors
Time interval for the timer is not setted correctly
The timer is not started in correct timing
The heater is not switched off at correct timing
Faults in Primitive Object
Faults in product behavioral specification

Figure 3: Fault analysis of "*microwave oven*" using HSFTA

**Deployment to Primitive Object level**

This stage is carried out with conventional FTA.

Fig.3 shows a example of fault analysis of "*microwave oven*" using HSFTA. A dangerous situation, *outbreak of fire*, is deployed. The upper Fig. 3 shows a result of conventional FTA.

At this stage, all factors must be deployed to the following three kinds of items:

•*faults of Primitive Object*, •*physical faults*, •*unexpected human behaviors*.

**Deployment to** *mode* **and** *Method* **of Primitive Object**

At this stage, fault factors are deployed to *modes* and *Methods* of Primitive Objects and their relations. Designers enumerate more primitive factors that cause *Faults of Primitive Object* and analyze the relation between factors.

Until the following factors appear, deployments are repeated:

- incorrect operations of Primitive Object,
- destruction of software, *ROM*, *RAM*, by external factors,
- unexpected human behaviors, and
- fatigue of hardware by physical factors.

The lower Fig. 3 shows a result of HSFTA. This is a deployment result of "Trouble of Timer Object". The *most primitive factors* correspond to shaded boxes in these figures. The *most primitive factor* is a factor which can not be deployed any more.

The factors that cause fault of "Timer Object" are analyzed as follows.

**Faults in a product specification**

The timer cannot control the heating time. Furthermore it is deployed:

* *the timer does not start counting down when the heater is switched on,*
* *the heater is not turned off when the countdown of the timer has come to zero.*

These are the cases where software design faults exist in a product behavioral specification that controls "Timer Object" and "Heater Object".

**Faults in Primitive Objects**

User cannot set up the time interval at the timer correctly or the timer can not count down correctly. These faults may occurred when there are bugs in "Timer Object" implementation.

**Software destroyed**

External factors disturb a product in spite of a specification correctly implemented.

When the fault analysis has been completed, designers can define assertions to avoid *most primitive faults*. These assertions are invariant properties of a product specification.

### 3.1.2 Translating assertions into logical forms

In order to verify whether safety assertions hold in a product specification, safety assertions must be defined as formal notations. Safety assertions can be represented as constraints on plural *Primitive Objects*.

In order to avoid the factor "overheating situation" shown in Fig. 3, there are several assertions on Primitive Objects in product behavioral specification. The assertions are derived from the *primitive faults* of Fig. 3.

Following are some examples: *Constraints* on "Heater Object" and "Timer Object" controlling the heating time are:

(1)*whenever the heater is switched on, the timer should start immediately or it has already started,*

(2)*while mode of "Heater Object" is on, the timer should be checked whether the countdown of the timer has come to zero,*

(3)*whenever the countdown of the timer come to zero, the heater should be switched off immediately.*

These constraints must be translated into logical formulas. Our approach represents constraints as the relations on Primitive Objects using followings.

(1) Heater[ON] $\Longrightarrow$ Timer[START] $\vee$ Timer(counting)

(2) Heater(on) $\Longrightarrow$ Timer[OVER]

(3) Timer[OVER] $\Longrightarrow$ Heater[OFF].

When all constraints hold in a product behavioral specification invariably, software design faults are avoided and safety of software is assured.

The following are notations to represent the constraints of Primitive Objects.

**Constraints of Software Primitive Objects**

- **Mode Condition**: Object$_\alpha$($mode_A$)
  means that *mode* of Object$_\alpha$ is $mode_A$.
- **Method Condition**: Object$_\beta$[Method$_B$]
  means that Method$_B$ of Object$_\beta$ is executed immediately.
- $A \Longrightarrow B$

states that "whenever $A$ succeeds, $B$ must hold invariably".

Both sides of formula can be consists of *Mode Conditions*, *Method Conditions* and three connectives such as $\vee$, $\wedge$, $\neg$.

### 3.2 Verification procedure

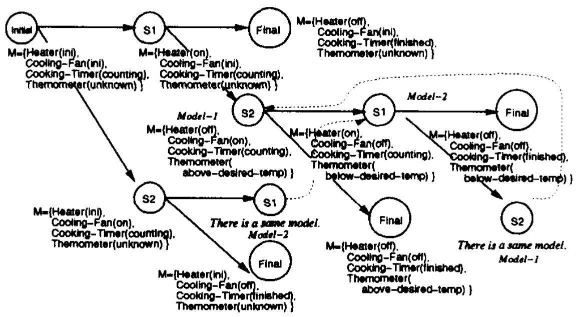
Our verification method computes a *reachability graph* of a product behavioral specification and analyzes statically whether this graph can hold safety constraints derived from HSFTA.

#### 3.2.1 Computing Reachability Graph

Considering a product behavioral specification as a directed graph, a *reachability graph* is generated by simulating State Transition Diagram from a initial state. This graph shows the sequences of product's activities.

A reachability graph in Fig. 4 is computed from a product behavioral specification in Fig. 2.

A reachability graph is composed of *nodes* and directed *branches*. A branch is a connection between adjacent nodes. Each *node* has a *model*. *Model* is a set of *mode* of Primitive Objects. Each branch corresponds to a transition in a product behavioral specification. One of the state's transitions is activated, new *node* is generated. The values of the *model* determine which of the current state's transitions is enabled.



Initial
S1
Final
M={Heater(off), Cooling-Fan(ini), Cooking-Timer(finished), Thermometer(unknown) }
M={Heater(ini), Cooling-Fan(ini), Cooking-Timer(counting), Thermometer(unknown) }
M={Heater(on), Cooling-Fan(ini), Cooking-Timer(counting), Thermometer(unknown) }
Model-1
S2
S1
Model-2
Final
M={Heater(off), Cooling-Fan(on), Cooking-Timer(counting), Thermometer( above-desired-temp) }
M={Heater(on), Cooling-Fan(off), Cooking-Timer(counting), Thermometer( below-desired-temp) }
M={Heater(off), Cooling-Fan(off), Cooking-Timer(finished), Thermometer( below-desired-temp) }
S2
S1
Final
S2
M={Heater(ini), Cooling-Fan(on), Cooking-Timer(counting), Thermometer(unknown) }
There is a same model.
Model-2
M={Heater(off), Cooling-Fan(off), Cooking-Timer(finished), Thermometer( above-desired-temp) }
There is a same model.
Model-1
Final
M={Heater(ini), Cooling-Fan(off), Cooking-Timer(finished), Thermometer(unknown) }

Figure 4: Reachability Graph for Fig. 2

#### 3.2.2 Checking the validity of Transitions

During computing this graph, our method checks this graph whether satisfies safety constraints.

The checker examines the following points:

- the relations between values of a current node's *model* and *events*,
- the relations between a new current node's *model* and a previous node's *model*.

When the checker determines that formulas are true invariably, then a safety assertion holds in a reachability graph and also in a product behavioral graph.

#### 3.2.3 Verification results

**Fault Sequence**

When this verification method detects an assertion which can not hold in a reachability graph, by tracing tree from *root node* to *fault branch & node* in a reachability graph, designers can localize the sequence of transitions in a product specification that may cause a fault.

**Fault Level**

Verifying all of assertions similarly, a set of assertions which can not hold in a product specification is derived. Using this set and the information of HSFTA, our method identifies *Fault Level*. *Fault Level* means most undesirable situations of a product that may be caused by plural *primitive faults*. When it has been detected that several assertions can not hold, primitive faults that correspond to these assertions are identified. These primitive faults are propagated to upper layer faults in HSFTA-diagram. Whether occurrence of factors will be propagated to upper layer factor depends on the logical combination of lower factors.

## 4 Discussion

We have implemented a verification system based on our method. In order to evaluate our method, we have applied this system to the software development of *microwave oven*.

We have described 41 sheets of State Transition Diagrams. The total number of states were about 500. The total number of transitions were about 1,000. As a result of applying *Specification Reduction Algorithm*, our verification system has generated the reachability tree having about 140,000 nodes.

We have analyzed the detected all kinds of errors in our product. Our verification system could detect 65% of errors in the product behavioral specification automatically. This value is a result of having verified not only safety assertions but also other kinds of assertions defined using our logical notations.

Considering safety of a product totally, there are several kinds of faults in addition to design faults of software: software destruction, hardware trouble, data destruction, etc. External factors cause these faults. It is necessary to prepare *Fail-safe* for these faults. When a *physical fault* factor is derived from HSFTA, designers decide how to handle the *fail-safe requirement* in hardware and software. When the *fail-safe requirements* in software are translated into the assertions using our logical formulas, those assertions can be verified automatically.

## 5 Conclusion

We have developed an automatic verifying approach for *software product specifications*. We have focused on verification of *safety requirements*. For this purpose, following items are provided.

First, we have shown an outline of software design method. Second, we have provided HSFTA method that derives safety assertions of specification.We have provided the logical notations to represent safety assertions. Third, we have provided the procedure for verifying whether a specification can satisfy safety assertions. This procedure has two steps: computing reachability graph and checking transitions. Fourth, we have provided the procedure for localizing design faults of specification and identifying the Fault Level. Finally, the possibilities and the limitations of our approach have been inspected through implementation and application.

**Reference**

[1] J-C.Laprie and B. Littlewood, *Probabilitic Assessment of Safety-Critical Software: WHY AND HOW?* CACM, Vol.35, No.2, Feb. 1992.