REAL-TIME OBJECT-ORIENTED SPECIFICATION
AND VERIFICATION

SATOSHI YAMANE
Deptartment of Computer Science, Shimane University
1060 Nishikawatu, Matsue City, ZIP/Zone 690, JAPAN
E-mail: yamane@cis.shimane-u.ac.jp

ABSTRACT

It is useful to introduce object-orientation into analysis and design methodologies for hard real-time software. But in the current object-oriented analysis and design methodology for hard real-time software, there is no methodology including formal verification of both dynamic properties and structural properties. In this paper, we propose specification and verification method as follows.

- Specification language consists of OMT syntax and timed automaton.
- Structural properties between object model and dynamic model, and functional model are verified.
- Dynamic properties such as event sequences and fairness are verified.

We have developed a verification system and shown this method effective.

1. Introduction

Hard real-time software such as automobile and airplane control software consists of many concurrent processes and many states, and especially timing constraints are strict. It is necessary to model, specify and verify hard real-time software including timing constraints. As noted in Kavi, there are many development methods and specification methods such as real-time structured analysis and Petri nets, automaton, process algebras, temporal logic, timed Petri nets, timed automaton, timed process algebra and real-time temporal logic. But there are many problems such as lack of modeling techniques, understandability and documentability.

On the other hand, object-oriented method is promising in database and information processing systems because of its modeling technique, information hiding, polymorphism and inheritance. Moreover, in hard real-time software, object-orientation is effective as follows.

- Encapsulation: As hard real-time software has many processes and states, a system specification has a huge number of states. But as system states are divided into local states of each object, these can be encapsulated. From this point of view, a large system can be specified.
- Concurrency: Hard real-time software consists of many concurrent processes. As each object is computed by message communication and it is
logically autonomous, objects are suitable for concurrency.

There are many object-oriented analysis and design methods for real-time software such as OMT (Object Modeling Technique\textsuperscript{17}), Schlaer/Mellor method\textsuperscript{20}, Booch method\textsuperscript{4}, Buhr method\textsuperscript{6}, ROOM (Real-Time Object-Oriented Modeling)\textsuperscript{18}, and Objectchart\textsuperscript{7}. And there are many object-oriented specification languages such as TROLL\textsuperscript{12}, OBLOG\textsuperscript{19}, TLOOM\textsuperscript{8} and OS\textsuperscript{5} based on OMT.

There are various problems with these methods and specification languages:

1. There is no formal specification and verification method in OMT, Schlaer/Mellor, Booch, and Buhr.
2. It is possible to specify timing conditions in ROOM, but there is no formal verification method.
3. It is possible to specify dynamic model and event traces in Objectchart, but there is no method to verify whether dynamic model satisfies event traces or not.
4. There is no method in order to verify structural properties such as relations between object model, dynamic model and functional model.
5. It is impossible to specify generalization, inheritance and timing conditions in TROLL and OBLOG.

In this paper, we propose a specification and verification method as follows.

- Specification language is the extended OMT by formalizing timing constraints.
- We can specify relations between three models and verify whether specification satisfies relations or not. This is the structural verification.
- It is possible to specify event traces (which objects send or receive) and verify whether specification satisfies event traces or not. This is the dynamic verification.

The structure of the paper is as follows. An object-oriented specification language for hard real-time software is proposed in section 2. The verification of a system structure is proposed in section 3. The verification of system dynamics is proposed in section 4. The verification system based on the method is shown in section 5. The conclusion is in section 6.

2. Object-Oriented Specification Language

This section outlines an object-oriented specification language. It is assumed that class is equal to object and object has no creation/destruction. The syntax of the specification language is based on OMT extending dynamic model with timed automaton\textsuperscript{1}. Specification language consists of twelve definition parts such as class name, attributes, operations, as shown in Figure 1.
specification language := (object definition part)

object definition part := Class name definition part ;
  Attribute definition part ;
  Operation definition part ;
  Association definition part ;
  [ State definition part ;
  Event definition part ;
  Initial state definition part ;
  Acceptance state definition part ;
  State transition definition part ;
  Data definition part ;
  Store definition part ;
  Function definition part ; ]

Classend ;

(1) Class name definition part := Class class name ;
(2) Attribute definition part := Attribute attribute name : type definition
    { attribute name : type definition } ;
    where type definition := integer | real | bool | char | set(a:b) | range(a:b)
(3) Operation definition part := Operation operation name [ , operation name ] ;
(4) Association definition part := Association association : association name[attribute][class name]
    { association : association name[attribute][class name] } ;
    where association := aggregation | inheritance | refer/use
(5) State definition part := State state name ( , state name ) ;
(6) Event definition part := Event event name : type definition [state name]
    { event name : type definition [state name] } ;
    where class name which inputs/outputs events
(7) Initial state definition part := state name ( , state name ) ;
(8) Acceptance state definition part := state name ( , state name ) ;
(9) State transition definition part := Transition state name (event, timing constraint/function
    next state name
    { state name (event , timing constraint/function next state name ) ;
    where timing constraint & := x < d | x > d | ¬ & 1 & 1 ∧ 2 | x = 0
    x : clock variable d : time constant
(10) Data definition part := Dataflow data name : type definition [class name]
    { data name : type definition [class name] } ;
(11) Store definition part := Store store name : type definition [class name]
    { store name : type definition [class name] } ;
(12) Function definition part := Function data name | store name ( , data name | store name)
    → function ( , data name | store name )
    { data name | store name ( , data name | store name)
    → function ( , data name | store name )

where [ ] and { } I are BNF notation

Fig. 1 Definition of syntax of specification language
State transition definition part(9) has the following features.

**Definition 1 (state transition definition part)** State transition definition part is defined by six tuples \((\Sigma, S, S_0, C, E, F)\) where

- \(\Sigma\) : a finite set of events (events are represented by object: event)
- \(S\) : a finite set of states
- \(S_0 \subseteq S\) : a finite set of start states
- \(C\) : a finite set of clocks
- \(E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)\) : a set of transitions
- \(F \subseteq S\) : a set of accepting states
- \(\Phi(C)\) represents timing constraints \(\delta\) of clock \(C\), and is recursively defined by a set \(X\) of clocks and a time constant \(D\) as follows.
  \[\delta := X \leq D \text{ or } D \leq X \text{ or } -\delta \text{ or } \delta_1 \land \delta_2\]

The timed automaton is extended by object: event instead of the event. This makes sending or receiving object clear.

A run \(r\) is accepting if some state from set \(F\) repeats infinitely often along \(r\).

3. The Verification of Structural Properties

3.1. The Structural Properties of Objects

In this paper, we assume that there are certain relations between object model, dynamic model and functional model as shown in Figure 2. These relations are classified into three categories: (1) case of aggregation, (2) case of inheritance, and (3) case of refer/use as follows.

1. **Case of aggregation** Dynamic model is mapped into the statechart\(^9\) and functional model is mapped into the concurrent program.

**Definition 2 (aggregation)**

- **Object model is defined as follows.**
  - Class \(O1\);
  - association aggregation : \(O2, O3\);

- **In dynamic model, it is as follows.**
  - Here \(\psi(S1)=\text{AND}\) and \(\rho(S1)=\{S2, S3\}\).
  - \(S1 = S2 \parallel S3\)
  - Namely, dynamic model is the statechart consisting of \(S2\) and \(S3\).

- **In functional model, it is as follows.**
  - Function \(F1 = F2 \parallel F3\)
Fig. 2 mapping relations between three models
– Namely, functional model is the concurrent program.

• where
  – $\psi$: type function $\mathcal{S} \rightarrow \{\text{AND, OR}\}$ (AND: concurrent, OR: hierarchical)
  – $\rho$: hierarchical function $\mathcal{S} \rightarrow 2^\mathcal{S}$ determines the substates of each state.
  – $\parallel$: concurrent operator
  – $\mathcal{S}_i$: the dynamic model of class $O_i$
  – $F_i$: the functional model of class $O_i$ ($i=1,2,3$)

(2) **Case of inheritance** Dynamic model is mapped into the statechart and functional model is mapped into the differencial program.

**Definition 3 (inheritance)**

• Object model is defined as follows.
  – Class $O_1$;
  – association inheritance: $O_2, O_3$;

• In dynamic model, it is as follows.
  – Here is $\psi(S_1) = \text{OR}$ and $\rho(S_1) = \{S_2, S_3\}$.
  – $S_1 = S_2 \text{ OR } S_3$
  – Namely, dynamic model is the statechart consisting of $S_2$ and $S_3$.

• In functional model, it is as follows.
  – Function $O_2 = F_2 \Delta F_1$;
  – Function $O_3 = F_3 \Delta F_1$;
  – Namely, the function of $O_2$ consists of $F_1$ and $F_2$. $F_2$ is the differencial program.

where $\Delta$: a differencial operator.

(3) **Case of refer/use** Dynamic model is mapped into the sequential automaton and functional model is mapped into the sequential program.

**Definition 4 (refer/use)**

• Object model is defined as follows.
  – Class $O_2$;
  – association refer/use: $O_1, O_3$;

• The class $O_1$ is as follows.
  – Class $O_1$;
  – association refer/use: $O_2$;
  – event $I_1: \text{bool}(O_2)$;
- or
- dataflow D1 : integer(O2);
- (store)

- The class O3 is as follows.

- Class O3;
- association refer/use : O2;
- event I3 : bool(O2);
- or
- dataflow D3 : integer(O2);
- (store).

3.2. The Verification Method of Structural Properties

According to the structural properties, we verify whether the dynamic model and functional model of a specification satisfy the structural properties in Figure 2. The verification system accepts as input the specification and the structural properties, and outputs the result as satisfiable/unsatisfiable as shown in Figure 3. Designers produce specification and structural properties specification. The structural properties specification must correspond to relations between object model, dynamic model and functional model.

Fig. 3 Verification method of structural property
4. The Verification of Dynamic Properties

4.1. The Dynamic Properties of Objects

The dynamic properties of objects represent the event traces between objects by their cooperation behavior. The event traces are the object scenarios and verification properties. We specify the event traces using deterministic timed Muller automaton\(^1\), which is closed under complementation. And the timed automaton is extended by object: event instead of the event. The timed automaton is formally defined as follows.

**Definition 5 (deterministic timed Muller automaton (DTMA))**

The deterministic timed Muller automaton is defined by a six tuple \((\Sigma, S, S_0, C, E, F)\) where

- \(\Sigma: a \) finite set of events (object: event)
- \(S: a \) finite set of states
- \(S_0 \subseteq S: a \) finite set of start states
- \(C: a \) finite set of clocks
- \(E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C): a \) set of transitions
- \(F \subseteq 2^S: an \) acceptance family
- \(\Phi(C)\) represents timing constraints \(\delta\) of clock \(C\), and is recursively defined by a set \(X\) of clocks and a time constant \(D\) as follows.

\[
\delta := X \leq D \text{ or } D \leq X \text{ or } -\delta \text{ or } \delta_1 \land \delta_2
\]

A run \(r\) is accepting if a run \(r \in F\).

4.2. The Verification Method of Dynamic Properties

If both the dynamic specification and verification specification are described in the automaton, the verification problem reduces to language inclusion problem\(^1\). The notion of verification is shown in Figure 4.

**Language inclusion problem** Let \(M_1\) be the timed automaton of dynamics specification and \(L(M_1)\) the language accepted by \(M_1\). Let \(M_2\) be the timed automaton of verification specification and \(L(M_2)\) the language accepted by \(M_2\). Then the language inclusion problem \(L(M_1) \subseteq L(M_2)\) is equal to testing whether

\[
L(M_1) \cap \text{the complementation of } L(M_2) = \varnothing.
\]

The complement of \(L(M_2)\) should be recognized by DTMA. \(\blacksquare\)

In order to reduce the verification problem to language inclusion problem, it is necessary to generate a "system dynamic model" from each dynamic model.
From structural properties of objects, we generate the system dynamic model. The generation rules are as shown in Figure 2.

1. Case of aggregation: the parallel composition of each dynamic model.
2. Case of inheritance: the hierarchical composition of each dynamic model.
3. Case of use/refer: the sequential composition of each dynamic object.

The system dynamic model is formally defined as follows.

**Definition 6 (system dynamic model)** The system dynamic model is defined by eight tuples $(\Sigma), S, S_0, C, E, F, \psi, \rho$ where

- $\Sigma$: a finite set of events (object: event)
- $S$: a finite set of states
- $S_0 \subseteq S$: a finite set of start states
- $C$: a finite set of clocks
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$: a set of transitions
- $F \subseteq S$: a set of accepting states
- $\Phi(C)$ represents timing constraints $\delta$ of clock $C$, and is recursively defined by a set $X$ of clocks and a time constant $D$ as follows.
  $$\delta := X \leq D \text{ or } D \leq X \text{ or } -\delta \text{ or } \delta_1 \land \delta_2$$
- $\psi$: type function $S \rightarrow \{\text{AND}, \text{OR}\}$ (AND: concurrent, OR: hierarchical)
- $\rho$: hierarchical function $S \rightarrow 2^S$ determines the substates of each state.

A run $r$ is accepting if some state from set $F$ repeats infinitely often along $r$.

The system dynamic model can be transformed into a timed automaton as follows.
Definition 7 The system dynamic model is modeled by an AND/OR tree, which has a unique root. In order to transform the system dynamic model into a timed automaton, following operations are applied from the root to leaves.

1. If $\psi(S)=$AND at level $i$, produce new states by intersecting $\rho(S)$ at level $i+1$.
2. If $\psi(S)=$OR at $i$ level, produce $\rho(S)$ at level $i+1$.

Next, we define the verification algorithm as follows.

The verification algorithm The verification algorithm such as language inclusion problem “$L(M1) \cap \text{the complementation of } L(M2) = \emptyset$” reduces to checking the emptiness of “$L(M1) \cap \text{the complementation of } L(M2)$”. The verification algorithm consists of the following steps.

1. Intersecting the automaton $L(M1)$ with the complementation of $L(M2)$.
2. Searching for a cycle meeting all the desired acceptance conditions by Tarjan’ depth-first search algorithm\(^{21}\), which is applied to sets of states.
3. If there is no acceptance cycle, $L(M1) \cap \text{the complementation of } L(M2) = \emptyset$ is satisfiable.
4. If there is a cycle, we check timing conditions of the cycle by the reachability analysis of DBM(Difference Bounds Matrices)\(^{8,2,16}\). If the reachability analysis is not satisfiable, $L(M1) \cap \text{the complementation of } L(M2) = \emptyset$ is satisfiable.

Here, we define DBM and the reachability analysis as follows.

Definition 8 (DBM(Difference Bounds Matrices)) DBM is a matrix consisting of inequalities, which represent timing constraints. The inequality is defined as $t_i \cdot t_j \leq d_{ij}$ where $t_i$ and $t_j$ are a set of clock variables, and $d_{ij}$ is a clock constant (integer). The $(i,j)$-factor of DBMs is $d_{ij}$. The fictitious clock variable $t_0$ is equal to 0.

Next, we define the reachability analysis.

The reachability analysis The reachability analysis involves the following steps:

1. Constructing DBM per each state.
2. Constructing a canonical DBM by Floyd-Warshall algorithm.
3. Analizing forward reachability of state transitions by the intersection of DBMs of transition relations.
4. If there is a negative-cost cycle, there is no state transition because of unsatisfiable timing conditions. If there are no negative-cost cycles, there are state transitions.
Next, we explain the verification example by the language inclusion algorithm.

**Example 1** The specification and verification property are given by Figure 5(1)(2). The verification property means “infinitely often a → infinitely often not a”, namely “infinitely often a → infinitely often c”. The accepting states of the specification are \{S0, S1\}. The acceptance family of the verification property is \{\{P1\}, \{P0, P1\}\}, and its complementation is \{\{P0\}\}.

Next, we intersect the specification with the complementation of the verification property as shown in Figure 5(3). The acceptance family of the intersection automaton is \{\{S0 × P0\}, \{S1 × P0\}\}. We found the following acceptance cycle by depth-first search:

\[(S0,P0) \rightarrow (S1,P0) \rightarrow (S0,P0) \rightarrow \ldots\]

Next, we check the reachability by DBMs as shown in Figure 5(4). Firstly, we produce DBMs and canonical DBMs of the acceptance cycle. Secondly, we produce the intersection of DBMs about state transition relations. Thirdly, we check whether there is a negative cost cycle in the canonical intersection of DBMs. In this case, there is an acceptance cycle because there are no negative cost cycles of DBMs. For this example, the specification does not satisfy the verification property.  

5. Example Specification and Verification

5.1. The Explanation of Problem

Assume that an automobile consists of accel and engine, transmission, brake as shown in Figure 6. Automobile communicates with the driver and external environment. Automobile is an aggregation of accel and engine, transmission, brake. Brake is inherited to normal brake and ABS (Antilock Brake System). Normal brake is operated by the driver, and ABS is computer-controlled in order not to lock wheels. Both normal brake and ABS control oil pressure and operate brake. For example, accel value is data flow, and accel on is control flow as shown in Figure 6.

5.2. Analysis and Specification

Automobile consists of accel, engine, transmission and brake. Each component is modeled as objects. A typical event sequence between objects is shown in Figure 7. For example, a driver sends [ignition on] to engine, accel sends [valve open] to engine, and engine sends [output rotation] to environment.
(1) specification  
\(s_0 \xrightarrow{a,x=0} s_1\)  \[c, x \leq 5\]

(2) verification property specification

At \((S1,P0)\), from \(x = 0, y \geq 0, y \geq x\)

DBM \(D1 = 0^0\)  
canonical DBM = 000

At \((S0,P0)\), from \(x \leq 5, y \geq 0, y \geq x\)

DBM \(D2 = 5^0\)  
canonical DBM = 500

At \((S1,P0) \Rightarrow (S0,P0)\), from \(x \leq 5, y \geq 0, y \geq x\) after \(x = 0, y \geq 0, y \geq x\).

intersection \(D1 \cap D2\)  
canonical DBM = 550

(3) product timed automaton

(4) Example of reachability analysis by DBM

Fig. 5 Verification example of language inclusion problem
5.2.1. The object model

Object model consists of object name and attribute (internal states), operation (methods). Associations consist of aggregation, inheritance, and refer/use. Associations can have attributes. For example, the association [control valve] has the attribute [valve value].

Object model has the following features.

1. System is an aggregation of accel, engine, transmission and brake.
2. Normal brake and ABS are an inheritance of brake.
3. Accel and engine are combined by refer/use association of [control valve].
   Transmision and brake are combined by refer/use association of [admit gear R]. Data flow and control flow are communicated by associations.

The object model of automobile is shown in Figure 8.
<table>
<thead>
<tr>
<th>driver</th>
<th>accel</th>
<th>engine</th>
<th>transmission</th>
<th>brake</th>
<th>environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td></td>
<td>ignition on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>brake on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>gear change</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>brake off</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>accel on</td>
<td></td>
<td>valve open</td>
<td></td>
<td></td>
<td>output rotation</td>
</tr>
<tr>
<td>accel off</td>
<td></td>
<td>valve close</td>
<td></td>
<td></td>
<td>output rotation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>brake on</td>
<td></td>
<td></td>
<td>car stop</td>
</tr>
</tbody>
</table>

Fig. 7 Event sequence diagram
Fig. 8 Object model
5.2.2. The dynamic model

The dynamic model is represented by a timed statechart. Each sequential set of states is represented by a rectangle (for example, accel and engine). Each hierarchical set of states is represented by rectangles in a dotted line (for example, brake on). Each state is represented by an ellipse (for example, accel off and accel increase). State transitions (edges) are represented by [input event, timing constraints/output event] format. In order to make the diagram clear, an underline is added to external events and a star is added to events between objects; no symbols are added to internal events. As soon as brake locks, ABS operates. But if ABS operates for more than 7 times, ABS has trouble.

The dynamic model has the following features.

1. Concurrent statechart consists of accel, engine, transmission and brake.
2. Hierarchical statechart consists of normal brake and ABS.
3. Transmission and brake communicate by [brake active] event.

The dynamic model is shown in Figure 9.

5.2.3. The functional model

Functional model is represented by dataflow model. Each process is represented by a circle (for example, activate accel on). Each store is represented by two lines (accel value). Each dataflow is represented a line (for example, valve value). Each control flow is represented by a dotted line (for example, valve open). Each functional model of components is represented by a rectangle.

Functional model has the following features.

1. Accel, engine, transmission and brake behave concurrently.
2. Normal brake and ABS are described as differential programs for inheritance.

In the functional model, underlines and stars are added to data and events in order to make the scope of control clear. As in real-time structured analysis\textsuperscript{10}, the following rule makes computational semantics clear.

1. An active process continues to execute during state transition.
2. As the process continues to be activated, data flow and control flow (event) continue to flow.
3. In general, a process can not output events; but the process outputs events only by data condition.

The functional model is shown in Figure 10.
where
external event: event name
event from another object: event name ✱
internal event: event name
ENV: external environment

Fig. 9 Dynamic model
Fig. 10 Function model
5.3. The Verification System

The verification system based on this method is shown in Figure 11. It consists of a verifier and compiler, and is implemented in 4Kstep C language. The compiler accepts as input specifications described in programming language format, and outputs three models described in list structures. The verifier accepts as input three models, dynamic property specification, and structural property specification. The verifier tests whether system specification satisfies property specifications or not.
Class accel;
Attribute
    accel value : real,
Operation
    operate accel,
Aggregation
    refer/use:
        control valve[valve value](engine);
State
    accel off, accel increase, . . . ;
Event
    accel on : bool(environment),
    accel keep : bool(environment),
    accel off : bool(environment),
car stop : bool(environment),
Initial
    accel off;
Acceptance
    accel off;
    accel increase;
    accel invariance;
    accel decrease;
Transition
    accel off  ENV : accel on/operate accel on  accel increase;
    .
    .
    accel invariance  ENV : accel off/operate accel off  accel decrease;
    accel decrease  ENV : car stop/;
Dataflow
    valve value : real(engine),
    valve open : bool(engine),
    valve keep : bool(engine),
    valve close : bool(engine);
Store
    accel value : real(environment);
Function
    accel value  operate accel on  valve open, valve value,
    accel value  operate accel keep  valve keep, valve value,
    accel value  operate accel off  valve off, valve value,
Classend;

Fig.12 Example of input data into compiler

The input data into the compiler is represented by the programming language format as shown in Figure 12.

5.4. Verification Example

5.4.1. The verification of a structural property

We show the verification of brake. The normal brake and ABS inherits properties from brake. We can specify the following structural properties:

1. brake = normal brake OR ABS
2. Function normal brake = normal brake ∆ control oil pressure
3. Function ABS = ABS ∆ control oil pressure
Next, we verify the consistency between the specification and the structural properties.

In the case of (1), as the structural property (1) exists in the specification, (1) is consistent with the property. In the case of (2),

\[ \text{slip ratio, brake value} \rightarrow \text{normal brake} \rightarrow \text{oil value, brake active, lock} \]

is defined in the specification. By oil value,

\[ \text{oil value} \rightarrow \text{control oil pressure} \rightarrow \text{control power} \]

becomes activated. As the function of normal brake consists of normal brake and control oil pressure, the specification is consistent with normal brake \(\Delta\) control oil pressure. (3) is the same as (2). The structural verification realized by the above method is shown in Figure 13.

---

**specification**

```
Class brake;
Association inheritance : normal brake, ABS;
:
brace on=normal brake O R ABS;
:
Function
oil value \(\rightarrow\) control oil pressure \(\rightarrow\) control power;
Class normal brake;
:
Function
slip ratio, brake value \(\rightarrow\) normal brake \(\rightarrow\) oil value, brake active, lock;
```

---

**verification**

**structural property specification**

```
brace=normal brake O R ABS
```

Function normal brake=normal brake \(\Delta\) control oil pressure
Function ABS=ABS \(\Delta\) control oil pressure

---

Fig. 13 Example of verification property of structural property
5.4.2. The verification of dynamic property

We explain the verification of dynamic properties in Figure 14. The verification properties is shown in Figure 14(1)(2).

1. A part of event traces between brake and ENV.
2. Strong fairness property.

Figure 14(1) consists of three states(S0, S1, S2). It is called time response property. Figure 14(2) consists of four states(S0, S1, S2, S3). It is called strong fairness such as infinitely often ENV:accel on $\rightarrow$ infinitely often ENV:brake on.

These properties are verified by the verification system using the language inclusion algorithm. In general, the language inclusion algorithm is PSPACE-
complete\textsuperscript{11,15}. But we apply Tarjan’s algorithm to sets of states rather than individual states\textsuperscript{21}. We could easily verify the specifications. The verification system terminated giving the correct answer “yes” (for this example, the running times were approximately 5-60 seconds on a SUN4/IP machine).

6. Conclusion

In this paper, an object-oriented verification method for hard real-time software is proposed, and we have realized the verification system. It is shown effective by example of automobile. Main features of the method are as follows.

1. We verify whether a system specification satisfies event traces and timing constraints.
2. We verify whether the relations of models satisfy structural properties.

A real problem is that complex processes are multi-tasking and each process has timing conditions. If we want to apply this method to complex processes, each process can be specified as an object and then the method can be applied. Though there is a state explosion problem, it can be overcome by symbolic techniques. We are now developing a real-time symbolic verification technique.

7. References