数理解析研究所講究録
第 618 巻 1987 年 110-129

Monitoring Ada® Tasking Programs Correctly

程 京徳　　荒木 啓二郎　　牛島 和夫
Jingde CHENG , Keijiro ARAKI , and Kazuo USHIJIMA

Department of Computer Science and Communication Engineering
Kyushu University
Hakozaki, Higashi-ku, Fukuoka 812, Japan

Abstract A new correctness concept, called partial-order preserving property, for event-driven execution monitoring of Ada tasking programs is presented. By using this concept, we can describe whether or not the tasking behavior of monitored Ada programs refrains from interference by monitoring actions of an event-driven execution monitor. In this paper, we define the equivalence of dynamic concurrent structures with respect to Ada program transformation, and propose this equivalence as a partial-order preserving criterion of the program transformation used in a preprocessor of an event-driven execution monitor of Ada tasking programs. The equivalence is formally based on the lattice of dynamic concurrent structure of Ada programs which provides an abstraction of the tasking behavior of Ada programs in terms of task interactions.

1. Introduction

In a software development environment, monitoring the behavior of programs is very important in order to verify if a monitored program satisfies both functional and performance requirements [Fairley-80, Plattner-81, Snodgrass-84, Taylor-85]. This is particularly critical in concurrent applications, where it is often hard to discover errors and faults (e.g., time-dependent errors, deadlocks, and livelocks) which can be traced by accurate execution monitoring [German-84, Helmbold-85a, 85b, LeBlanc-85, LeDoux-85, Maio-85].

Execution monitoring is one of the facilities to be supported by an Ada programming support environment (APSE) [DoD-80, Fairley-80, Taylor-85]. On the basis of this requirement, we are developing an event-driven execution monitor for Ada tasking programs. Our execution monitor is a testing and/or debugging tool. It monitors the execution of a target Ada tasking program, reports information about dynamic tasking behavior of the program, detects tasking communication deadlocks (if any) during execution of the program, saves traces of tasking behavior of the program, analyses timing of task interactions, and answers the queries about saved tasking behavior of the program to users.

® Ada is a registered trademark of the U. S. Government (Ada Joint Progam Office).

When we develop the event-driven execution monitor for Ada tasking programs, it is indispensable to make certain that the monitor should always report correctly the information about tasking behavior of target programs.

In our approach, execution monitoring is achieved through interaction between a target Ada tasking program and the execution monitor running in parallel. In order to let the monitor follow what is happening in the program, the program must tell the monitor what its tasks are doing. To this end, the target program undergoes a series of transformations at source code level. These transformations introduce the monitor implemented as an Ada task, assign a unique identifier to each task, and insert calls to the monitor. So the monitor can follow the task interactions of the program at run time, collects information about tasking behavior from the program, and detects deadlocks.

The major factors affecting correctness of information reported by the execution monitor are as follows:

1) Because of interference by monitoring actions of the monitor, tasking behavior of the transformed program P' may be so different from that of the original program P that the constraints on the ordering of events of P are changed. This problem arises not only in monitoring Ada tasking programs, but also in determining the correctness of Ada-to-Ada program transformation;

2) Because of fault of transforming and/or monitoring algorithms, the "tasking behavior of the target program" reported by the monitor may be incomplete or false even if the real tasking behavior of the transformed program P' preserves constraints on the ordering of tasking events of the original program P.

This paper discusses how to solve the first problem. The second problem relates to the "completeness" and "pertinence" of the execution monitor which have been discussed in detail in [Plattner-81]. These concepts can also be applied to develop an event-driven execution monitor of Ada tasking programs and further discussion is beyond the scope of this paper.

Several execution monitors of Ada programs have been developed for various aims [German-84, Helmbold-85a,85b, LeDoux-85, Maio-85], but, these literatures have not discussed how to preserve constraints on the ordering of events of the original program during execution monitoring. So they fail to prove formally the correctness of the transforming and/or monitoring algorithms used in the execution monitors.

In this paper, we present a new correctness concept, called partial-order preserving property, for event-driven execution monitoring of Ada tasking programs. By using this concept, we can describe whether or not the behavior of monitored Ada tasking programs refrains from interference by monitoring actions of the event-driven execution monitor. In section 2, we define the tasking state space of Ada programs for determining the domain of our discussion. In section 3, we formalize the task interactions of Ada programs by using the lattice theory in order to provide an abstraction of the tasking behavior of Ada programs in terms of task interactions. In section 4, we define the equivalence of dynamic concurrent structures with respect to Ada program transformation and propose it as a partial-order preserving criterion of program transformation used in the preprocessor of an event-driven execution monitor of Ada tasking programs. In section 5, we give some simple examples. Concluding remarks are given in section 6.

For the following discussion, knowledge of the semantics of Ada tasking [DoD-83] are prerequisites.

2. Tasking State Space of Ada Programs

The primary mechanism for Ada task interactions is the rendezvous form of remote procedure call. Because communication and synchronization among tasks using message passing are more general than those using shared variables, we concentrate attention on only task interactions in terms of rendezvous even though the Ada tasks may interact by using shared variables. It is assumed that the tasks of all Ada tasking programs referred in this paper interact by using only rendezvous form. Thus, the tasking behavior of such Ada programs is determined by the input and the rendezvous among tasks.

Because we would like to be able to understand an Ada tasking program in terms of its component tasks and their interaction, without regard for how they are executed, we make no assumption about execution rates of concurrently executing tasks, except the finite progress assumption (i.e., rates of tasks all are positive) .

On the other hand, the main program of an Ada program acts as if called by some environment task [DoD-83]. In this paper, such an environment task is called the *main task*. It is assumed that the main task starts its execution at the elaboration of all library units needed by the main program, and terminates its execution at the termination of all tasks and the main program.

According to the semantics of Ada tasking [DoD-83], a task may be in any one of the following states :

1) *Starting activation* : The elaboration of the declarative part of the task is started.
2) *Activating* : The declarative part of the task is being elaborated.
3) Activated : The declarative part of the task has been elaborated.
4) *Executing* : A statement in the body of the task or a subprogram called by the task is being executed.
5) *Delay* : The task is being delayed as a result of execution of a delay statement in its own body.
6) *Entry calling* : The task has issued an entry call (simple or conditional or timed) to another task but this entry call has not yet been accepted.
7) *Accepting* : The task is waiting at an accept statement for any corresponding entry call.
8) *Selective waiting* : The task is waiting at a selective wait for any one of its opened select alternatives to be selected.

9) *Starting block activation* : The elaboration of the declarative part of a block statement in the body of the task is started; or the elaboration of the declarative part of a subprogram called by the task is started.

10) *Block Activating* : The declarative part of a block statement in the body of the task is being elaborated; or the declarative part of a subprogram called by the task is being elaborated.

11) *Block Activated* : The declarative part of a block statement in the body of the task has been elaborated; or the declarative part of a subprogram called by the task has been elaborated.

12) *Block completed* : The execution of a block statement in the body of the task has been completed; or the execution of a subprogram called by the task has been completed.

13) *Block termination waiting* : The task is waiting at the end of a block statement in its own body for all tasks dependent on that block statement to terminate; or the task is waiting at the end of a subprogram called by itself for all tasks dependent on that subprogram to terminate.

14) *Block terminated* : The execution of a block statement in the body of the task has been completed, and all dependent tasks of the block statement have terminated; or the execution of a subprogram called by the task has been completed, and all dependent tasks of the subprogram have terminated.

15) *Abnormal* : The task is abnormal as the result of the execution of an abort statement.

16) *Completed* : The execution of statements in the body of the task has been completed.

17) *Termination waiting* : The task is waiting at the end of its own body for all dependent tasks to terminate.

18) *Terminated* : The execution of statements in the body of the task has been completed, and all dependent tasks of the task have terminated.

19) *Rendezvous* : The task has issued an entry call to another task which has accepted this entry call; or the task has accepted an entry call issued by another task.

20) *Suspended by rendezvous* : The task has issued an entry call which has been accepted by the called task, but the execution of the corresponding accept statement has not yet been completed.

21) *Continue* : The task which has issued an entry call resumes its execution as the result of the completion of the corresponding rendezvous; or the task has completed the execution of an accept statement and continues its execution.

A sequence of states of a task from "Starting activation" to "Terminated" is called a *life cycle* of the task.

Definition 2.1 A *dynamic concurrent state* of a task is specified by a quintuplet (T, t, s, e, m), where, T is a task identifier by which the task can be uniquely identified during the execution of the program;

t is time* when the state of the task T is changed;

s is a state of the task T;

e is a communicating entry between the task T and the other task, and is defined if and only if s is in any one of the following states :

Entry calling,
Accepting,
Rendezvous,
Suspended by rendezvous, and
Continue,

otherwise, e is not defined and denoted by $\perp$;

m is a set of messages which are passed between the task T and the other task during their rendezvous, and is defined if and only if s is in any one of the following states :

Entry calling,
Accepting,
Rendezvous, and
Continue,

otherwise, m is not defined and denoted by $\perp$. □

Definition 2.2 A *tasking state space* TSS(P, I) created by an execution of Ada tasking program P with input I is a set of dynamic concurrent states passed by any task of P as a result of the execution of P which reads I as input. □

Note that repeated executions of an Ada tasking program P with the same input I may create different tasking state spaces even without regard for the time in dynamic concurrent states. Because P may contain nondeterministic statements, repeated executions of P with the same input I may execute different paths in P.

We also note that TSS(P, I) will be infinite when the execution of P does not halt.

3. The Dynamic Concurrent Structure of Ada Programs

We now formalize the task interactions of Ada tasking programs for providing an abstraction of the tasking behavior of Ada programs to discuss partial-order preserving property of event-driven execution monitoring.

A relation $\leqq$ on the set A is called a partial order if it satisfies the three conditions [Birkhoff-61] :

R (Reflexivity): For every a in A, $a \leqq a$.

A (Antisymmetry): For every a, a' in A, if both $a \leqq a'$ and $a' \leqq a$ hold, then $a = a'$.

* This time may be physical time in an interleaved implementation of Ada, or may be virtual time in a distributed implementation of Ada [Lamport-78, Jefferson-85].

T (Transitivity): For every a, a' , a" in A, if both a $\leqq$ a' and a' $\leqq$ a" hold, then a $\leqq$ a".

If $\leqq$ is a partial order on A, we call the pair (A, $\leqq$) a poset (short for partially ordered set) [Birkhoff-61].

Two elements a, a' of a poset (A, $\leqq$) are called to be incomparable with respect to $\leqq$ if neither a $\leqq$ a' nor a' $\leqq$ a [Birkhoff-61].

We assume that change of state of a task is an event in an execution of the task.

There may exist a binary relation "happened after" between two states of a tasking state space TSS(P, I) which describes the occurrence order of two states according to Ada tasking semantics. For example, a tasks must be created after the start of execution of its master, and must terminate before the termination of execution of its master.

The relation "happened after" is only a partial order of the states in the tasking state space TSS(P, I) because it is sometimes that no order between two states is determined by the semantics of program P. This partial order can be viewed to be a minimum constraints on the occurrence order of tasking events during the execution of program P, and is determined only by the semantics of program P.

The relation "happened after" can be formally defined as follows.

Definition 3.1 Given any Ada tasking program P, for any two tasks A and B during an execution of the P, the task A is called the *parent* of the task B, and the task B is called a *child* of the task A if and only if they satisfy any one of the following conditions :

1) The task A is the master on which the task B depends directly.

2) The task A is the master on which the task B depends indirectly, but there exists no task which is the master of the task B and depends on the task A.

3) The task A is the main task, and the master on which the task B depends directly is a library package.

4) The task A is the main task, and the master on which the task B depends indirectly is a library package, but there exists no task that is the master of the task B and depends on the library package. □

Definition 3.2 Given any TSS(P, I), a binary relation DS : TSS(P, I) $\rightarrow$ TSS(P, I) is defined as follows :

For any two elements $S_1 = (T_1, t_1, s_1, e_1, m_1)$ and $S_2 = (T_2, t_2, s_2, e_2, m_2)$ in the TSS(P, I), $(S_1, S_2) \in$ DS if and only if S_1 and S_2 satisfy any one of the following conditions :

1) $T_2 = T_1$,
$t_2 = \min\{ t \mid (T, t, s, e, m) \in \text{TSS(P, I)} \wedge t \geqq t_1 \wedge T = T_1 \wedge s \neq s_1 \}$,
$s_2 \neq s_1$.

2) T_2 is a child of T_1,
$t_2 = \min\{ t \mid (T, t, s, e, m) \in \text{TSS(P, I)} \wedge t \geqq t_1 \wedge (T \text{ is a child of } T_1) \wedge$
$s = \text{Starting activation}\}$,

s_2 = Starting activation,
s_1 = Starting activation $\vee$ s_1 = Starting block activation.

3) T_2 is a parent of T_1,
$t_2 = \min\{t \mid (T, t, s, e, m) \in \text{TSS}(P, I) \wedge t \geqq t_1 \wedge (T \text{ is the parent of } T_1) \wedge (s = \text{Activated} \vee s = \text{Block activated})\}$,
s_2 = Activated $\vee$ s_2 = Block activated, s_1 = Activated.

4) T_2 is the parent of T_1,
$t_2 = \min\{t \mid (T, t, s, e, m) \in \text{TSS}(P, I) \wedge t \geqq t_1 \wedge (T \text{ is the parent of } T_1) \wedge (s = \text{Terminated} \vee s = \text{Block terminated})\}$,
s_2 = Terminated $\vee$ s_2 = Block terminated, s_1 = Terminated.

5) An entry of T_2 is called by T_1, $t_2 = t_1$, $s_2 = s_1$ = Rendezvous, $e_2 = e_1$.

6) T_2 called an entry of T_1, $t_2 = t_1$, $s_2 = s_1$ = Continue, $e_2 = e_1$.

If $(S_1, S_2) \in DS$, then S_2 is called a *direct successor* of S_1. □

Definition 3.3 Given any TSS(P, I), the transitive reflexive closure of DS on the TSS(P, I) is called *tasking partial ordering relation* of Ada program P and is denoted by $\leqq_{TPO}$ for DS^*. □

Lemma 3.1 Given any TSS(P, I), then $\leqq_{TPO}$ is a partial order on the TSS(P, I), i.e., (TSS(P, I), $\leqq_{TPO}$) is a poset. □

Let (A, $\leqq$) be a poset. An element of A, denoted by **0**, is called the least element of A if $\mathbf{0} \leqq a$ for every a in A. An element of A, denoted by **1**, is called the greatest element of A if $a \leqq \mathbf{1}$ for every a in A. [Birkhoff-61]

Lemma 3.2 Given any (TSS(P, I), $\leqq_{TPO}$), then it has the least element **0** given by $\mathbf{0} = (M, t_0, \text{Starting activation}, \bot, \bot)$; if (TSS(P, I), $\leqq_{TPO}$) is finite, then it also has the greatest element **1** given by $\mathbf{1} = (M, t, \text{Terminated}, \bot, \bot)$; where M is the identifier of the main task of P. □

A poset (A, $\leqq$) is called a chain if either $a \leqq a'$ or $a' \leqq a$ for every a, a' in A [Birkhoff-61].

Lemma 3.3 Given any (TSS(P, I), $\leqq_{TPO}$), then any subset of the (TSS(P, I), $\leqq_{TPO}$) in which any elements have the same task identifier is a chain. □

Lemma 3.1 ~ Lemma 3.3 can be proved by **Definition 3.1 ~ 3.3.**

Let (A, $\leqq$) be a poset. Let A' be a subset of A. An element b of A is called an lower bound (l.b.) for A' if $b \leqq a$ for every a in A', b is called greatest lower bound (g.l.b.) for A', if $b' \leqq b$ for every l.b. b' for A'. An element b of A is called an upper bound (u.p.) for A' if $a \leqq b$ for every a in A', b is called least upper bound (l.u.p.) for A' if $b \leqq b'$ for every u.b. b' for A'. We denote g.l.b. and l.u.p. for A' by $\cap$ A' and $\cup$ A' respectively. If A' has only two elements, we write $a_1 \cap a_2$ (read : a_1 meet a_2) and $a_1 \cup a_2$ (read : a_1 join a_2), respectively, for $\cap \{a_1, a_2\}$ and $\cup \{a_1, a_2\}$ [Birkhoff-61].

A meet-semilattice is a poset L any two of whose elements have a g.l.b.; a join-semilattice is a poset L any two of whose elements have a l.u.b.; a lattice is a poset

L which is both a meet-semilattice and a join-semilattice; a complete lattice is a poset L in which every subset has both a g.l.b. and a l.u.b. [Szasz-63].

Theorem 3.1 Given any Ada program P, then any (TSS(P, I), $\leqq_{TPO}$) is a meet-semilattice; if (TSS(P, I), $\leqq_{TPO}$) is finite, then it is not only a lattice but also a complete lattice. □

Theorem 3.1 can be proved by **Lemma 3.1 ~ 3.3** and Ada tasking semantics; further discussion is given in the appendix of this paper.

Lattice (TSS(P, I), $\leqq_{TPO}$) (both semilattice and lattice) is called the *lattice of dynamic concurrent structure of Ada programs*; it provides an abstraction of the tasking behavior of Ada programs in terms of task interactions.

We may represent a poset (A, $\leqq$) by Hesse diagrams. In where, the nodes represent the element of A, there is a path from a to a' which goes entirely in the downward direction if and only if a $\leqq$ a'.

An example of the lattice of dynamic concurrent structure of a simple Ada program P (see Fig.1 (a)) is shown in Fig. 1 (b) by using Hasse diagrams.

```
procedure M is
  task type ADD is
    entry ADD_I ( X, Y : in INTEGER; Z : out INTEGER );
    entry ADD_R ( X, Y : in REAL; Z : out REAL );
  end ADD;
  type TP is access ADD;
  task body ADD is
  begin
    loop
      select
        accept ADD_I ( X, Y : in INTEGER; Z : out INTEGER ) do Z := X + Y; end;
      or
        accept ADD_R ( X, Y : in REAL; Z : out REAL ) do Z := X + Y; end;
      or
        terminate;
      end select;
    end loop;
  end ADD;
  A, B, C, S : INTEGER;   D, E, F : REAL;   T1 : ADD;   T2 : TP;
begin
  A := 1;   B := 2;   D := 3.3;   E := 4.4;   T1.ADD_I ( A, B, C ) ;
  T2 := new ADD;   T2.all.ADD_R ( D, E, F ) ;
  S := C + INTEGER ( F ) ;
end M;
```

Fig. 1 (a) A simple Ada tasking program P

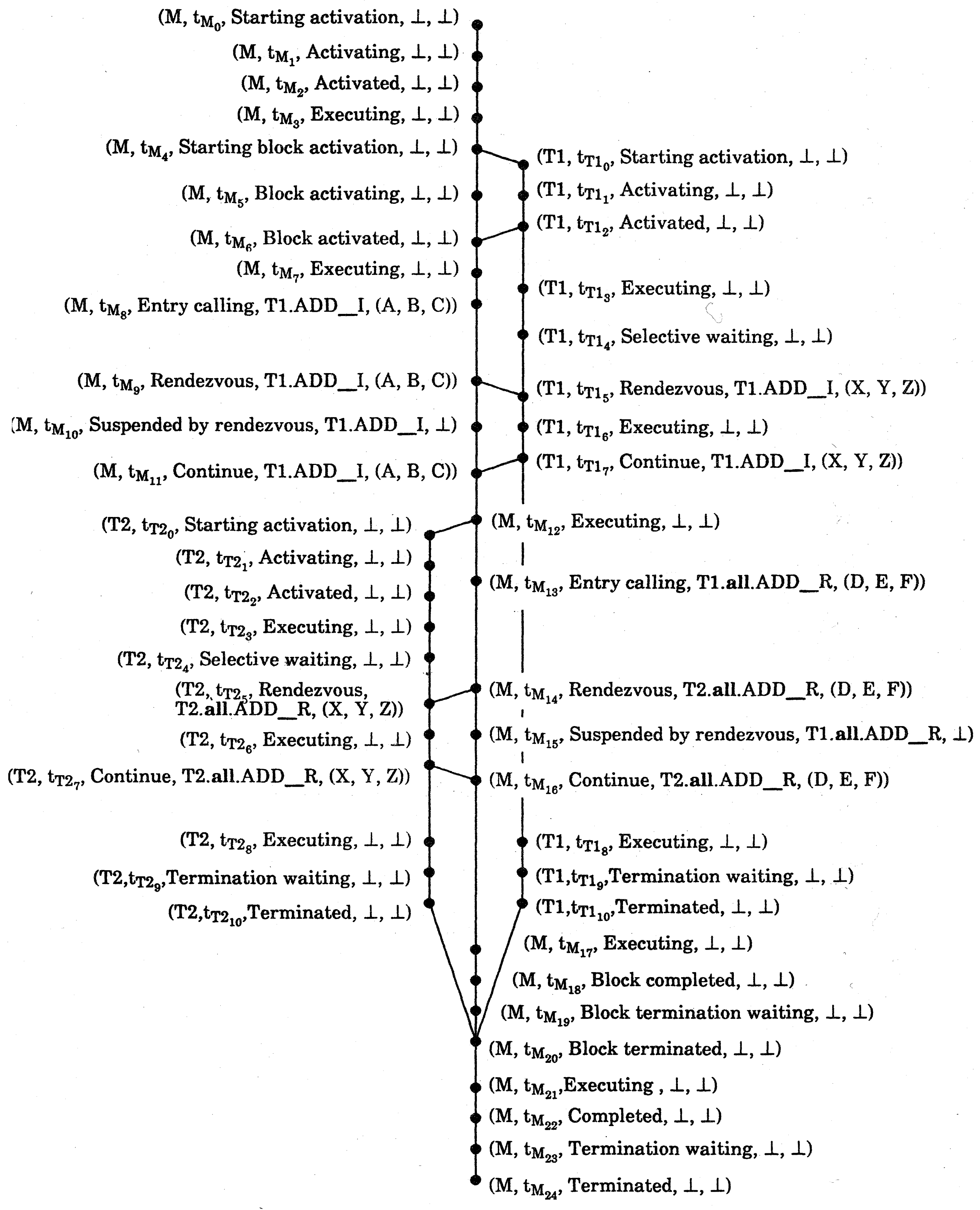

(M, t_{M_0}, Starting activation, ⊥, ⊥)
(M, t_{M_1}, Activating, ⊥, ⊥)
(M, t_{M_2}, Activated, ⊥, ⊥)
(M, t_{M_3}, Executing, ⊥, ⊥)
(M, t_{M_4}, Starting block activation, ⊥, ⊥)
(T1, t_{T1_0}, Starting activation, ⊥, ⊥)
(T1, t_{T1_1}, Activating, ⊥, ⊥)
(M, t_{M_5}, Block activating, ⊥, ⊥)
(T1, t_{T1_2}, Activated, ⊥, ⊥)
(M, t_{M_6}, Block activated, ⊥, ⊥)
(M, t_{M_7}, Executing, ⊥, ⊥)
(T1, t_{T1_3}, Executing, ⊥, ⊥)
(M, t_{M_8}, Entry calling, T1.ADD_I, (A, B, C))
(T1, t_{T1_4}, Selective waiting, ⊥, ⊥)
(M, t_{M_9}, Rendezvous, T1.ADD_I, (A, B, C))
(T1, t_{T1_5}, Rendezvous, T1.ADD_I, (X, Y, Z))
(M, $t_{M_{10}}$, Suspended by rendezvous, T1.ADD_I, ⊥)
(T1, t_{T1_6}, Executing, ⊥, ⊥)
(M, $t_{M_{11}}$, Continue, T1.ADD_I, (A, B, C))
(T1, t_{T1_7}, Continue, T1.ADD_I, (X, Y, Z))
(T2, t_{T2_0}, Starting activation, ⊥, ⊥)
(M, $t_{M_{12}}$, Executing, ⊥, ⊥)
(T2, t_{T2_1}, Activating, ⊥, ⊥)
(T2, t_{T2_2}, Activated, ⊥, ⊥)
(M, $t_{M_{13}}$, Entry calling, T1.all.ADD_R, (D, E, F))
(T2, t_{T2_3}, Executing, ⊥, ⊥)
(T2, t_{T2_4}, Selective waiting, ⊥, ⊥)
(T2, t_{T2_5}, Rendezvous, T2.all.ADD_R, (X, Y, Z))
(M, $t_{M_{14}}$, Rendezvous, T2.all.ADD_R, (D, E, F))
(T2, t_{T2_6}, Executing, ⊥, ⊥)
(M, $t_{M_{15}}$, Suspended by rendezvous, T1.all.ADD_R, ⊥)
(T2, t_{T2_7}, Continue, T2.all.ADD_R, (X, Y, Z))
(M, $t_{M_{16}}$, Continue, T2.all.ADD_R, (D, E, F))
(T2, t_{T2_8}, Executing, ⊥, ⊥)
(T1, t_{T1_8}, Executing, ⊥, ⊥)
(T2,t_{T2_9},Termination waiting, ⊥, ⊥)
(T1,t_{T1_9},Termination waiting, ⊥, ⊥)
(T2,$t_{T2_{10}}$,Terminated, ⊥, ⊥)
(T1,$t_{T1_{10}}$,Terminated, ⊥, ⊥)
(M, $t_{M_{17}}$, Executing, ⊥, ⊥)
(M, $t_{M_{18}}$, Block completed, ⊥, ⊥)
(M, $t_{M_{19}}$, Block termination waiting, ⊥, ⊥)
(M, $t_{M_{20}}$, Block terminated, ⊥, ⊥)
(M, $t_{M_{21}}$,Executing , ⊥, ⊥)
(M, $t_{M_{22}}$, Completed, ⊥, ⊥)
(M, $t_{M_{23}}$, Termination waiting, ⊥, ⊥)
(M, $t_{M_{24}}$, Terminated, ⊥, ⊥)

Fig. 1 (b) Lattice of dynamic concurrent structrue of program P

4. The Equivalence of Dynamic Concurrent Structures on Ada Program Transformation

In our approach, it is obvious that the "tasking behavior of the target program" reported by the event-driven execution monitor is meaningful only if the partial order of tasking events during the execution of target program is not interfered by monitoring actions of the execution monitor. This means that the event-driven execution monitoring must preserve the partial order of tasking events which is determined only by the semantics of the target program.

We say that an event-driven execution monitoring is *partial-order preserving* if it preserves the partial order of tasking events which is determined only by the semantics of the target program.

We also say that an event-driven execution monitor of Ada tasking programs is *partial-order-transparent* if its execution monitoring is always partial-order preserving for any target Ada program. In other words, the existence of such a monitor is transparent for any target Ada program with respect to the partial order of tasking events during execution of the program which is determined only by the semantics of program.

On the basis of the lattice of dynamic concurrent structure of Ada programs, we can discuss the equivalence between a target Ada program P and a transformed Ada program P' with respect to preserving the partial order of tasking events during execution of P which is determined only by the semantics of P.

We assume that an Ada-to-Ada program transformation is a partial mapping between two sets of Ada programs, and can specified by a set of program transformation rules. A program transformation rule is also a partial mapping from a set of Ada programs to the other.

We write $\Phi : \mathbf{P}(\mathrm{Ada}) \rightarrow_R \mathbf{P}(\mathrm{Ada})$ for an Ada-to-Ada program transformation, where $\mathbf{P}(\mathrm{Ada})$ is a set of Ada programs and R is a set of Ada program transformation rules, R can be omitted when it need not be shown explicitly.

We now define a series of basic concepts which will be used in later discussions.

Definition 4.1 Let $(S_1, \leqq_1)$ and $(S_2, \leqq_2)$ be two posets.

1) A mapping $\Psi : S_1 \rightarrow S_2$ is called an order homomorphism or an isotone or an order preserving mapping if $s_1 \leqq_1 s_2$ then $\Psi(s_1) \leqq_2 \Psi(s_2)$ for any s_1, s_2 in S_1 [Birkhoff-61, Szasz-63].

2) A mapping $\Psi : S_1 \rightarrow S_2$ is called a *conservative homomorphism* or a *conservative mapping* if s_1 and s_2 are incomparable with respect to $\leqq_1$ then $\Psi(s_1)$ and $\Psi(s_2)$ are incomparable with respect to $\leqq_2$ for any s_1, s_2 in S_1 .

3) A mapping $\Psi : S_1 \rightarrow S_2$ is called a *partial-order homomorphism* or a *partial-order preserving mapping* if and only if it is both an order homomorphism and a conservative homomorphism. □

Definition 4.2 Let $(L_1, \leqq_1)$ and $(L_2, \leqq_2)$ be two meet-semilattices.

1) A mapping $\Psi : (L_1, \leqq_1) \to (L_2, \leqq_2)$ is called a meet-homomorphism if $\Psi(a \cap b) = \Psi(a) \cap \Psi(b)$ for any a, b in L_1 [Birkhoff-61, Szasz-63].

2) A meet-homomorphism is called a *meet-homomorphous one-to-one* if it is also one-to-one.

3) A meet-homomorphism is called a *meet-homomorphous onto* if it is also onto.

4) A meet-homomorphism is called a *meet-isomorphism* if it is both a meet-homomorphous one-to-one and a meet-homomorphous onto. □

Lemma 4.1 Let $\Psi : (L_1, \leqq_1) \to (L_2, \leqq_2)$ be a bijection. Then Ψ is a partial-order homomorphism if and only if it is a meet-isomorphism. □

The proof of **Lemma 4.1** is given in the appendix of this paper.

Based on **Lemma 4.1**, we have

Definition 4.3 A program transformation $\Phi : \mathbf{P}(\mathrm{Ada}) \to \mathbf{P}(\mathrm{Ada})$ is called an *equivalence transformation for dynamic concurrent structures* if and only if for every P in the $\mathbf{P}$(Ada) and any given input I there exists a meet-homomorphism $\Psi : (\mathrm{TSS}(P, I), \leqq_{TPO}) \to (\mathrm{TSS}(\Phi(P), I), \leqq_{TPO})$ which is a meet-isomorphism from $(\mathrm{TSS}(P, I), \leqq_{TPO})$ to $(\Psi(\mathrm{TSS}(P, I)), \leqq_{TPO})$. □

In fact, **Definition 4.3** means that an equivalence transformation $\Phi : \mathbf{P}(\mathrm{Ada}) \to \mathbf{P}(\mathrm{Ada})$ for dynamic concurrent structures preserves the partial order of tasking events of original program P in the transformed program $\Phi(P)$.

Note that the equivalence of dynamic concurrent structure defined by **Definition 4.3** describes only a correspondence between the original and the transformed programs in terms of abstract algebraic structure, and no concrete semantics of original programs such as the identifier or the tasking state of a task is considered in this equivalence.

We can consider a strong equivalence of dynamic concurrent structure which describes the complete tasking semantics of original programs as follows :

Definition 4.4 A program transformation $\Phi : \mathbf{P}(\mathrm{Ada}) \to \mathbf{P}(\mathrm{Ada})$ is called a *strong equivalence transformation for dynamic concurrent structures* if and only if for every P in the $\mathbf{P}$(Ada) and any given input I, there exists a meet-homomorphism $\Psi : (\mathrm{TSS}(P, I), \leqq_{TPO}) \to (\mathrm{TSS}(\Phi(P), I), \leqq_{TPO})$ which is a meet-isomorphism from $(\mathrm{TSS}(P, I), \leqq_{TPO})$ to $(\Psi(\mathrm{TSS}(P, I)), \leqq_{TPO})$ and satisfies the following condition :

For any $(T_1, t_1, s_1, e_1, m_1)$ in the $(\mathrm{TSS}(P, I), \leqq_{TPO})$, if $\Psi((T_1, t_1, s_1, e_1, m_1)) = (T_2, t_2, s_2, e_2, m_2)$ then $T_1 = T_2, s_1 = s_2, e_1 = e_2, m_1 = m_2$. □

A strong equivalence transformation for dynamic concurrent structures preserves the partial order of tasking events and the complete tasking semantics of original programs in the transformed programs. Therefore we have the following proposition.

Proposition If the program transformation used by an event-driven execution monitor of Ada tasking programs is a strong equivalent transformation for dynamic concurrent structures, then the event-driven execution monitor is partial-order-transparent. □

Definition 4.5 Let Φ : **P**(Ada) $\to_R$ **P**(Ada) be a program transformation. Then the program transformation rules of R are said to be *superposable* if they satisfy the following condition :

For any P in the **P**(Ada), there exists a sequence of indices $i_1, i_2, ..., i_n$ for which the following formula holds :

$\Phi(P) = r_{i_n}(... (r_{i_2}(r_{i_1}(P)))...)$, where, $r_k \in R$, $k = i_1, i_2, ..., i_n$. □

Lemma 4.2 Let $\Psi_1 : (L_1, \leqq_1) \to (L_2, \leqq_2)$ and $\Psi_2 : (L_2, \leqq_2) \to (L_3, \leqq_3)$ are two meet-homomorphisms, then their composite $\Psi_2 \cdot \Psi_1 : (L_1, \leqq_1) \to (L_3, \leqq_3)$ is also a meet-homomorphism. □

Theorem 4.1 Let Φ : **P**(Ada) $\to_R$ **P**(Ada) be a program transformation. If R satisfies the conditions :

1) The program transformation rules of R are superposable;

2) For any program transformation rule r in R, and for every P in **P**(Ada) and any given input I, there exists a meet-homomorphous Ψ_r : (TSS(P, I), $\leqq_{TPO}$) $\to$ (TSS(r(P), I), $\leqq_{TPO}$) which is a meet-isomorphism from (TSS(P, I), $\leqq_{TPO}$) to (Ψ_r(TSS(P, I)), $\leqq_{TPO}$),

then Φ is an equivalent transformation for dynamic concurrent structures.

If in addition to the above conditions R satisfies the following condition :

3) For every Ψ_r in the condition 2) and for any $(T_1, t_1, s_1, e_1, m_1)$ in the (TSS(P, I), $\leqq_{TPO}$), if $\Psi_r((T_1, t_1, s_1, e_1, m_1)) = (T_2, t_2, s_2, e_2, m_2)$ then $T_1 = T_2$, $s_1 = s_2$, $e_1 = e_2$, $m_1 = m_2$,

then Φ is a strong equivalent transformation for dynamic concurrent structures.□

Theorem 4.1 can be proved by **Definition 4.3 ~ 4.5** and **Lemma 4.2.**

Theorem 4.1 means that the equivalence problem of dynamic concurrent structure of a program transformation can be reduced into the meet-homomorphous problem of every transformation rule of the program transformation. In general, the latter problem is easier to solve than the former because it only needs to discuss local properties of target programs.

Consequently, for developing a partial-order-transparent event-driven execution monitor of Ada tasking programs on the basis of above discussion, we must do two things. The first is to design program transformation rules which are superposable (and of course, satisfy the requirements of execution monitoring), and the second is to check whether or not every program transformation rule of the execution monitor satisfies the conditions of **Theorem 4.1.**

5. Examples

We now give two examples. The first example is a program transformation which introduces an entry SET__ID for every task type of a target program to accept unique identifier, but it is not strongly equivalent for dynamic concurrent structures. In the second example a revised transformation is applied to the same target programs as in the first example. It is strongly equivalent for dynamic concurrent structures.

5.1 Program Transformation without Strong Equivalence

Let us consider the following transformation [German-84] . It consists of a pattern and a replacement, where, a numeral in square brackets, for instance [1], is a pattern variable, the program text covered by a pattern variable is copied into the replacement. The pattern variables may be restricted; i.e., for instance, [2-statements] can only match a sequence of statements.

pattern :

```
task type T is [1] ;
task body T is [2-decls] begin [2-statements] ; end ;
```

replacement :

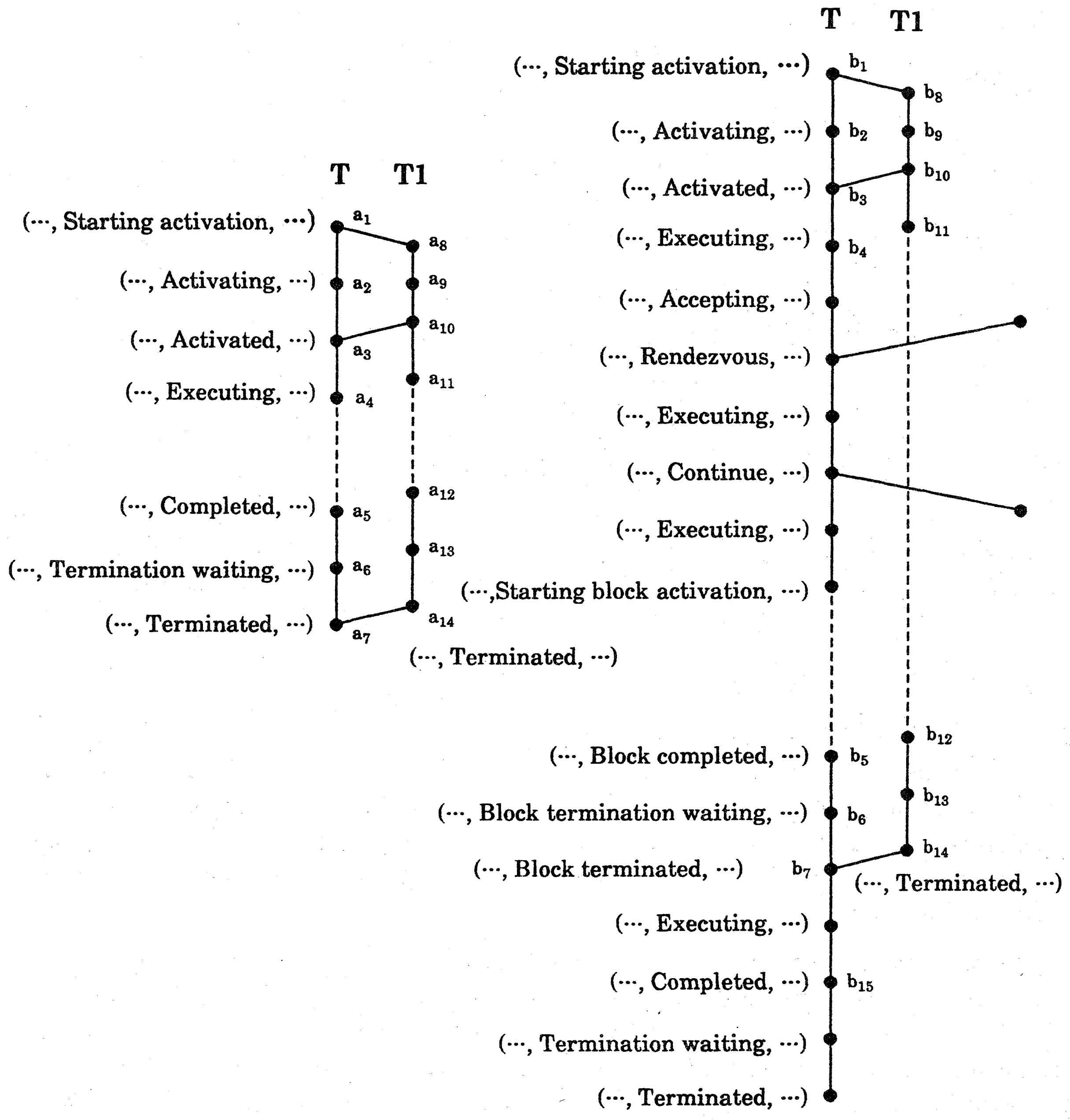
```
task type T is
   entry SET__ID ( N : in INTEGER ) ;
   [1] ;
task body T is
   ID : INTEGER ;
begin
   accept SET__ID ( N : in INTEGER ) do ID := N ; end ;
   declare
      [2-decls]
   begin
      [2-statements] ;
   end ;
end ;
```

This transformation moves the declarative part of the task body of a target program into an inner block in order to accept the identifier inside of the task body before elaborating the task's declarative part. But this transformation is not strongly equivalent for dynamic concurrent structures.

Let us assume that some task, for instance T1, is declared in the declarative part of task T, then the outline of lattice of dynamic concurrent structure of a target program and its transformed program is shown in Fig.2 (a) and Fig.2 (b) respectively.

Obviously, there exists no meet-isomomorphism that satisfies the conditions of **Definition 4.4** because elements a_5 and a_{14} are incomparable in lattice A but b_{15} and b_{14} are comparable in lattice B.

As a result of the transformation, because it is not strongly equivalent for dynamic concurrent structures, the transformed program may not preserve equivalence with respect to some tasking behavior of the target program such as exceptions raised during the elaboration of the declarative part of the task [German-84] .


T
T1
(…, Starting activation, …)
(…, Activating, …)
(…, Activated, …)
(…, Executing, …)
(…, Completed, …)
(…, Termination waiting, …)
(…, Terminated, …)
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14}
(…, Terminated, …)
T
T1
(…, Starting activation, …)
(…, Activating, …)
(…, Activated, …)
(…, Executing, …)
(…, Accepting, …)
(…, Rendezvous, …)
(…, Executing, …)
(…, Continue, …)
(…, Executing, …)
(…,Starting block activation, …)
(…, Block completed, …)
(…, Block termination waiting, …)
(…, Block terminated, …)
(…, Executing, …)
(…, Completed, …)
(…, Termination waiting, …)
(…, Terminated, …)
(…, Terminated, …)
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

(a) A : Lattice of target program (b) B : Lattice of transformed program

Fig. 2 An example of program transformation without strong equivalence

However, note that this transformation is equivalent for dynamic concurrent structures and a meet-isomomorphism which satisfies the conditions of Definition 4.3 has been shown in Fig.2 by the same indices of the corresponding elements.

5.2 Program Transformation with Strong Equivalence

The second example is a revision of the program transformation given by the first example. The revised transformation is shown below. Every task is assigned a unique identifier by initialization of task identifier ID by calling a function GET_TASK_ID.

pattern :

```
task type T is [1] ;
task body T is [2-decls] begin [2-statements] ; end ;
```

replacement :

```
task type T is [1] ;
task body T is
   ID : INTEGER := GET__TASK__ID ;
   [2-decls]
begin
   [2-statements] ;
end ;
```

where the function GET__TASK__ID is defined as follows :

```
function GET__TASK__ID return INTEGER is
   ID : INTEGER;
begin
   TASK__ID__MANAGER . GET__ID ( ID ) ; return ID ;
end ;
```

This transformation is strongly equivalent for dynamic concurrent structures. A meet-isomomorphism which satisfies the conditions of **Definition 4.4** is shown in Fig. 3 by the same indices of the corresponding elements.

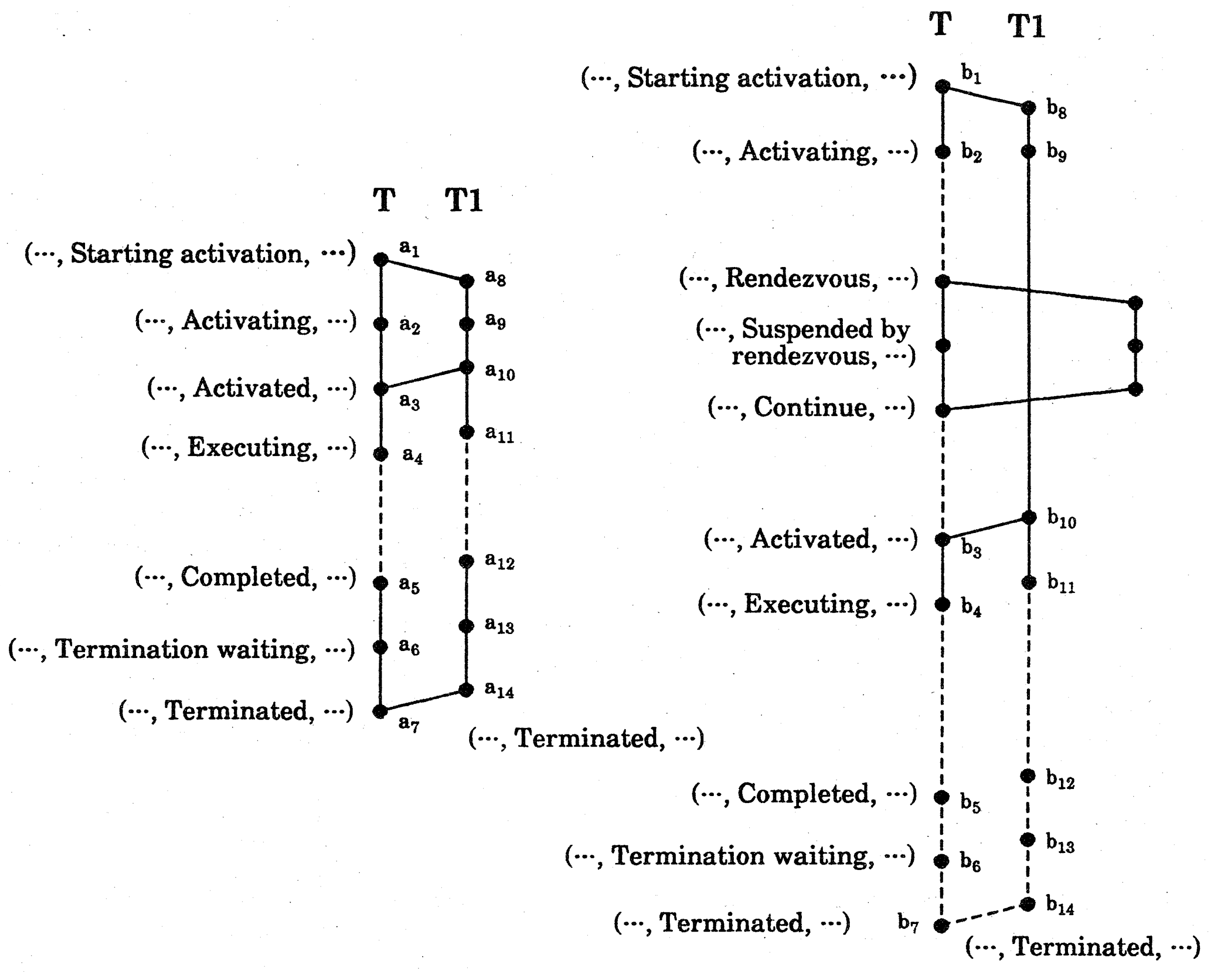
6. Conclusion

This paper has presented a new correctness concept, called partial-order preserving property, for event-driven execution monitoring of Ada tasking programs. By using this concept, we can describe whether or not the tasking behavior of monitored Ada programs refrains from interference of monitoring actions of the event-driven execution monitor.

We have presented an approach to solve the partial-order preserving problem which is important for both monitoring and transforming Ada tasking programs. The features of our approach are to abstract the tasking behavior of Ada programs in terms of task interactions by using lattice theory, and to discuss the correspondence between target programs and transformed programs in terms of abstract algebraic structure on the basis of this abstraction.

The equivalence of partial-order preserving property on transformation of concurrent programs is an important research area. In order to solve the partial-order preserving problem on monitoring Ada tasking programs, we have defined the equivalence of dynamic concurrent structures with respect to Ada program transformation. This equivalence can be used as a partial-order preserving criterion of program transformation used in a preprocessor of an event-driven execution monitor of Ada tasking programs. It can be easily extended to applications to other concurrent languages based on message-passing mechanism such as CSP [Cheng-86].

On the basis of the correctness concept presented in this paper, we have designed an event-driven execution monitor for Ada tasking programs and implemented its prototype system on an Ada processor named ADE [DGC-84]. The correctness of



(a) A : Lattice of target program (b) B : Lattice of transformed program

Fig. 3 An example of program transformation with strong equivalence

monitoring and/or transforming algorithms used by our event-driven execution monitor can be proved with the help of the strong equivalence of dynamic concurrent structures with respect to Ada program transformation.

References

[Birkhoff-61] Birkhoff,G. : Lattice Theory, Revised Edition, Am. Math. Soc., 1961.

[Cheng-86] Cheng,J., Araki,K. and Ushijima,K. : The Equivalence of Dynamic Concurrent Structures on Concurrent Program Transformation, Proc. 32th Annual Convention IPS Japan, 4C-7, pp.13-14, 1986 (in Japanese).

[DGC-84] Data General Corp. : Ada Development Environment (ADE) (AOS/VS) User's Manual, 1984.

[DoD-80] United States Department of Defense : "STONEMAN", Requirements for Ada Programming Support Environment, 1980.

[DoD-83] United States Department of Defense : Reference Manual for Ada Programming Language (ANSI/MIL-STD-1815A), Jan. 1983.

[Fairley-80] Fairley,R.E.: Ada Debugging and Testing Support Environments, ACM SIGPLAN Notices, Vol.15, No.11, pp.16-25, 1980.

[German-84] German,S.M. : Monitoring for Deadlock and Blocking in Ada Tasking, IEEE Trans. Softw. Eng., Vol.SE-10, No.6, pp.764-777, 1984.

[Helmbold-85a] Helmbold,D. and Luckham,D. : Debugging Ada Tasking Programs, IEEE Software, Vol.2, No.2, pp.47-57, 1985.

[Helmbold-85b] Helmbold,D. and Luckham,D.C. : Runtime Detection and Description of Deadness Errors in Ada Tasking, ACM Ada Letters, Vol.4, No.6, pp.60-72, 1985.

[Jefferson-85] Jefferson,D.R. : Virtual Time, ACM Trans. Program. Lang. Syst., Vol.7, No.3, pp.404-425, 1985.

[Lamport-78] Lamport,L. : Time, Clocks, and Ordering of Events in a Distributed System, Comm. ACM, Vol.21, No.7, pp.558-565, 1978.

[LeBlanc-85] LeBlanc,R.J. and Robbins,A.D. : Event-Driven Monitoring of Distributed Programs, Proc. of the 5th International Conference on Distributed Computing Systems, pp.515-522, 1985.

[LeDoux-85] LeDoux,C.H. and Parker,Jr.D.S. : Saving Traces for Ada Debugging, Ada in Use, pp.97-108, Proc. of the Ada International Conference, Paris 14-16 May 1985.

[Maio-85] Maio,A.D, Ceri,S. and Reghizzi,S.C. : Execution Monitoring and Debugging Tool for Ada Using Relational Algebra, Ada in Use, pp.109-123, Proc. of the Ada International Conference, Paris 14-16 May 1985.

[Plattner-81] Plattner,B. and Nievergelt,J. : Monitoring Program Execution: A Survey, IEEE Computer, Vol.14, No.11, pp.76-93, 1981.

[Snodgrass-84] Snodgrass,R. : Monitoring in a Software Development Environment: a Relational Approach, ACM SIGPLAN Notices Vol.19, No.5, pp.124-131, 1984.

[Szasz-63] Szasz,G. ; Introduction to Lattice Theory, Academic Press, 1963.

[Taylor-85] Taylor,R.N. and Standish,T.A. : Steps to an Advanced Ada Programming Environment, IEEE Trans. Softw. Eng., Vol.SE-11, No.3, pp.302-310, 1985.

Appendix

Proof of Theorem 3.1

For any $S_1 = (T_1, t_1, s_1, e_1, m_1)$, $S_2 = (T_2, t_2, s_2, e_2, m_2)$ in the $(TSS(P, I), \leqq_{TPO})$, if $T_1 = T_2$, then by **Lemma 3.3**, the following formula hold :

$$\text{g.l.b.}\ \{S_1, S_2\} = \begin{cases} S_1 & \text{when } S_2 \leqq_{TPO} S_1, \\ S_2 & \text{when } S_1 \leqq_{TPO} S_2. \end{cases}$$

If $T_1 \neq T_2$, then by **Lemma 3.2**, $\{S_1, S_2\}$ has at least one l.b. such as **0**.

If we assume that there exists no g.l.b. $\{S_1, S_2\}$, then $\{S_1, S_2\}$ must have two l.b.'s $b_1 = (T_1, t_{b_1}, s_{b_1}, e_{b_1}, m_{b_1})$ and $b_2 = (T_2, t_{b_2}, s_{b_2}, e_{b_2}, m_{b_2})$ which are incomparable because $\{S_1, S_2\}$ has at least one l.b.; i.e., $b_1 \leqq_{TPO} S_1$, $b_1 \leqq_{TPO} S_2$, $b_2 \leqq_{TPO} S_1$, $b_2 \leqq_{TPO} S_2$, but neither $b_1 \leqq_{TPO} b_2$ nor $b_2 \leqq_{TPO} b_1$.

According to **Definition 3.2**, the above assumption is true if and only if there are two elements $S_1' = (T_1, t_1', s_1', e_1', m_1')$ and $S_2' = (T_2, t_2', s_2', e_2', m_2')$ of the $(TSS(P, I), \leqq_{TPO})$ which satisfy the following conditions :

1) $b_1 \leqq_{TPO} S_1' \leqq_{TPO} S_1$, $b_2 \leqq_{TPO} S_2' \leqq_{TPO} S_2$;

2) $(b_1, S_2') \in$ **DS**, $(b_2, S_1') \in$ **DS**;

i.e., there is a valid combination of the conditions 2 ~ 6 of **Definition 3.2** for which $(b_1, S_2') \in$ **DS** and $(b_2, S_1') \in$ **DS** are to be true.

We now consider every combination of the condition 2 ~ 6 of **Definition 3.2** as follows; note that b_1 and b_2 are incomparable, and the number of combinations of those conditions is $\binom{5+2-1}{2} = 15$:

1) b_1 and S_2' satisfy the condition 2; b_2 and S_1' satisfy the condition 2 :
This means that T_2 is a child of T_1 and started activation at t_2'; and T_1 is a child of T_2 and started activation at t_1'.

2) b_1 and S_2' satisfy the condition 2; b_2 and S_1' satisfy the condition 3 :
This means that T_2 is a child of T_1 and started activation at t_2'; and T_1 is a parent of T_2 and T_2 is activated at t_{b_2}.

3) b_1 and S_2' satisfy the condition 2; b_2 and S_1' satisfy the condition 4 :
This means that T_2 is a child of T_1 and started activation at t_2'; and T_1 is a parent of T_2 and T_2 terminated at t_{b_2}.

4) b_1 and S_2' satisfy the condition 2; b_2 and S_1' satisfy the condition 5 :
This means that T_2 is a child of T_1 and started activation at t_2'; and T_2 called an entry of T_1 and this call is accepted by T_1 at t_1'.

5) b_1 and S_2' satisfy the condition 2; b_2 and S_1' satisfy the condition 6 :
This means that T_2 is a child of T_1 and started activation at t_2'; and T_1 called an entry of T_2 and corresponding rendezvous finished at t_{b_2}.

6) b_1 and S_2' satisfy the condition 3; b_2 and S_1' satisfy the condition 3 :
This means that T_2 is a parent of T_1 and T_1 started activation at t_{b_1}; and T_1 is a parent of T_2 and T_2 started activation at t_{b_2}.

7) b_1 and S_2' satisfy the condition 3; b_2 and S_1' satisfy the condition 4 :
This means that T_2 is a parent of T_1 and T_1 started activation at t_{b_1}; and T_1 is a parent of T_2 and T_2 terminated at t_{b_2}.

8) b_1 and S_2' satisfy the condition 3; b_2 and S_1' satisfy the condition 5 :
This means that T_2 is a parent of T_1 and T_1 started activation at t_{b_1}; and T_2 called an entry of T_1 and this call is accepted by T_1 at t_1'.

9) b_1 and S_2' satisfy the condition 3; b_2 and S_1' satisfy the condition 6 :
This means that T_2 is a parent of T_1 and T_1 started activation at t_{b_1}; and T_1 called an entry of T_2 and corresponding rendezvous finished at t_{b_2}.

10) b_1 and S_2' satisfy the condition 4; b_2 and S_1' satisfy the condition 4 :
This means that T_2 is a parent of T_1 and T_1 terminated at t_{b_1}; and T_1 is a parent of T_2 and T_2 terminated at t_{b_2}.

11) b_1 and S_2' satisfy the condition 4; b_2 and S_1' satisfy the condition 5 :
This means that T_2 is a parent of T_1 and T_1 terminated at t_{b_1}; and T_2 called an entry of T_1 and this call is accepted by T_1 at t_1'.

12) b_1 and S_2' satisfy the condition 4; b_2 and S_1' satisfy the condition 6 :
This means that T_2 is a parent of T_1 and T_1 terminated at t_{b_1}; and T_1 called an entry of T_2 and corresponding rendezvous finished at t_{b_2}.

13) b_1 and S_2' satisfy the condition 5; b_2 and S_1' satisfy the condition 5 :
This means that T_1 called an entry of T_2 and this call is accepted by T_2 at t_2'; and T_2 called an entry of T_1 and this call is accepted by T_1 at t_1'.

14) b_1 and S_2' satisfy the condition 5; b_2 and S_1' satisfy the condition 6 :
This means that T_1 called an entry of T_2 and this call is accepted by T_2 at t_2'; and T_1 called an entry of T_2 and corresponding rendezvous finished at t_{b_2}.

15) b_1 and S_2' satisfy the condition 6; b_2 and S_1' satisfy the condition 6 :
This means that T_2 called an entry of T_1 and corresponding rendezvous finished at t_{b_1}; and T_1 called an entry of T_2 and corresponding rendezvous finished at t_{b_2}.

However, any one of these 15 situations can not occur because of the semantics of Ada tasking [DoD-83]. This means that our assumption is not true. Therefore, there must exist g.l.b. $\{S_1, S_2\}$.

Consequently, for any S_1, S_2 in the (TSS(P, I), $\leqq_{TPO}$), $\{S_1, S_2\}$ has a g.l.b., i.e., the (TSS(P, I), $\leqq_{TPO}$) is a meet-semilattice.

On the other hand, if the (TSS(P, I), $\leqq_{TPO}$) is finite, then for any S_1, S_2 in the (TSS(P, I), $\leqq_{TPO}$), the discussion of l.u.b. $\{S_1, S_2\}$ holds as well as that of g.l.b. $\{S_1, S_2\}$. Therefore, the (TSS(P, I), $\leqq_{TPO}$) is a lattice. Because any finite lattice is complete lattice [Birkhoff-61, Szasz-63], the (TSS(P, I), $\leqq_{TPO}$) is a complete lattice. □

Proof of Lemma 4.1

Let $\Psi : (L_1, \leqq_1) \rightarrow (L_2, \leqq_2)$ be both a bijection and a partial-order homomorphism. Then for any a, b in L_1,

(1) $a \leqq_1 b$ iff $\Psi(a) \leqq_2 \Psi(b)$.

Suppose $a \cap b = c$, then $c \leqq_1 a$ and $c \leqq_1 b$. By (1),

(2) $\Psi(c) \leqq_2 \Psi(a)$, $\Psi(c) \leqq_2 \Psi(b)$, i.e., $\Psi(c) \leqq_2 \Psi(a) \cap \Psi(b)$.

Suppose $\Psi(a) \cap \Psi(b) = \Psi(d)$, then

(3) $\Psi(c) \leqq_2 \Psi(d)$, and

(4) $\Psi(d) \leqq_2 \Psi(a)$, $\Psi(d) \leqq_2 \Psi(b)$.

By (1) and (4),

(5) $d \leqq_1 a$, $d \leqq_1 b$, i.e., $d \leqq_1 a \cap b$, i.e., $d \leqq_1 c$.

Therefore, by (1),

(6) $\Psi(d) \leqq_2 \Psi(c)$.

By (3) and (6), $\Psi(d) = \Psi(c)$, i.e., $\Psi(a \cap b) = \Psi(a) \cap \Psi(b)$.

Consequently, Ψ is a meet-isomorphism.

Let $\Psi : (L_1, \leqq_1) \rightarrow (L_2, \leqq_2)$ be a meet-isomorphism. Then for any a, b in L_1,

(7) $\Psi(a \cap b) = \Psi(a) \cap \Psi(b)$.

Suppose $a \leqq_1 b$, then $a \cap b = a$, $\Psi(a \cap b) = \Psi(a)$. By (7)

(8) $\Psi(a) = \Psi(a) \cap \Psi(b)$, i.e., $\Psi(a) \leqq_2 \Psi(b)$.

On the other hand, suppose $\Psi(a) \leqq_2 \Psi(b)$, then $\Psi(a) \cap \Psi(b) = \Psi(a)$. By (7)

(9) $\Psi(a \cap b) = \Psi(a)$.

Therefore,

(10) $a \cap b = a$, i.e., $a \leqq_1 b$.

Consequently, Ψ is a partial-order homomorphism. □