Compositional Petri Net Models of Advanced Tasking in Ada-95[1]

Ravi K. Gedela, Sol M. Shatz and Haiping Xu
Concurrent Software Systems Lab
The University of Illinois at Chicago
Chicago, IL 60607 USA

Abstract

The Ada language has been designed to support development of concurrent and distributed software. While the Ada-83 standard defined the basic mechanisms of rendezvous-based tasking, the Ada-95 standard significantly extended this capability with the introduction of several advanced tasking constructs. We present and discuss formal models of these key tasking constructs using the Petri net model. We also provide some formal evaluation of the models using one particular net-based method, invariant analysis. The constructs considered are the asynchronous transfer of control, the protected object, and the requeue statement. By modeling these advanced Ada tasking constructs with Petri nets, we obtain compositional models of the constructs that are complementary to earlier work in net-based modeling of Ada tasking, both in terms of defining precise behavior for tasking semantics, and also in terms of providing support for automated analysis of concurrent software.

Keywords: Ada-95, Compositional Models, Concurrency, Petri Nets, Tasking

1. Introduction

As concurrent and distributed systems become more and more common, the development of such systems is creating new challenges in the development of software and for software engineering in general [1]. Designing languages that support concurrency is one such challenge. The Ada language has been designed to support development of concurrent and distributed software, especially for embedded, real-time systems. This support is provided by Ada's many constructs for multitasking. Unfortunately, the multitasking features can be difficult to understand, especially in terms of how various features interact. This is due in part to the fact that the semantics of Ada are "officially" defined in the Ada reference manual [2] in informal English; there is a lack of documented formal specifications for those tasking semantics. One direction for helping this situation is to use a structured and formal semantic model that explicitly provides precise and unambiguous definitions of behavior. One method for constructing real-time models of

[1] This work was supported in part by the U. S. National Science Foundation under Grant CCR-9321743 and by the U.S. Army Research Office under grant number DAAG55-98-1-0470.

Ada programs that support timing analysis was presented by Corbett [3]. That work used a hybrid automaton model to capture the execution times of tasks and common sources of run-time overhead. Since the focus of that work is on real-time modeling, it relies on various implementation-dependent features. In this paper, we consider implementation-independent "concurrency models" for the key tasking constructs of Ada-95. We use the Petri net model [4,5] since Petri nets not only provide a formal, mathematical-based semantic, but also provide the advantage of being quickly understood due to their simple construction and graphical depiction. In addition, Petri nets have a strong base of existing (and evolving) theory and development tools.

Petri nets can be effectively used to model tasking behavior since the net model explicitly supports features like concurrency, non-determinism, synchronization, and mutual exclusion. By modeling individual Ada tasking constructs with Petri nets, we can obtain compositional models of the constructs. These models define precise behavior and have the potential to support analysis of concurrent Ada programs. This is because Petri nets have well-known characteristics that allow one to identify properties of the model such as absence of deadlock, mutual exclusion violation, etc. Soon after Ada's first standard was adopted (the so-called Ada-83 standard [6]), Mandrioli, et al. [7] published Petri net models to define key Ada-83 tasking constructs. This work modeled the basic tasking constructs using timed Petri nets, and stressed the value of formal modeling of the semantics of the tasking constructs in Ada-83. Shatz et al. [8] also described Petri net models for Ada tasking constructs; but, the focus of that early modeling work was to provide a formal foundation for automated generation of general net models for tasking programs for the purpose of supporting net-based analysis of a program's tasking behavior [9,10]. This research showed significant promise toward automated analysis of non-trivial Ada tasking programs, especially for deadlock properties. It resulted in the development of a set of support tools and a comprehensive set of experimental results [9]. Some of this analysis work applied so-called net reduction to Petri net models in order to decrease the complexity of state-space analysis.

With the recent standardization of Ada-95 [2], we feel that it is valuable to now provide new Petri net models of the language primitives associated with Ada-95 tasking. The primitives considered are the asynchronous transfer of control, the protected object, and the requeue statement. These primitives have been shown to be valuable for implementing atomic actions to support software fault tolerance [11]. While net reductions similar to those mentioned before can also be applied to these new models, we do not automatically apply reductions in this paper because such reduced models would provide less semantic detail on the constructs being considered. Our first priority here is to demonstrate a set of compositional models that properly capture the intended tasking semantics; optimization of such models is a separate issue, but it can be addressed by reductions on the models we provide. In an earlier paper [12], we presented some Petri net models for Ada-95 tasking constructs. But the models in that work were not compositional, meaning that they were not constructed to support modular construction of systems that use the various tasking constructs. Furthermore, that earlier work did not present any formal evaluation of the models. In this paper, we present compositional models with formal evaluations based on invariant analysis.

The rest of the paper is organized as follows. Section 2 is a background section that briefly summarizes basic Ada tasking features and provides an overview of Petri nets. Readers who are already familiar with Ada tasking features and Petri net theory may skip this section. Section 3 first describes a traditional model for entry call and its evolved

version, which supports modeling advanced tasking constructs, it then discusses the construction and evaluation of Petri net models for the three tasking constructs that are newly defined in Ada-95: asynchronous transfer of control, protected object, and requeue statement. Section 4 provides a brief conclusion and summary of this work.

2. Background

2.1 Ada Tasking

Ada has been specifically designed to support concurrent and real time programming. The language has many constructs that facilitate concurrency, where concurrent activities are described by means of tasks. A task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks. A task unit is the basic unit for defining a task whose sequence of actions may be executed concurrently with those of other tasks. Such tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. Ada also provides a delay statement, which can be used to suspend, or delay, the execution of a task for a specific amount of time.

The interaction among tasks can be done in two fundamental ways [13]: by direct message communication and by indirect communication. Direct message communication is accomplished through a mechanism called *rendezvous*. In indirect communication, tasks interact through *protected object.* This feature was not supported in Ada-83; it provides exclusive write-only or concurrent read-only access to shared data. More detailed discussion of this feature is included in Section 3.2.2.

2.2 Overview of Petri Nets

Petri nets are a graphical and mathematical modeling tool. This model has shown value in describing and studying information systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic [4]. Petri nets allow one to build a model of a desired system, and analyze the model formally to study the behavior of the system. In this section we briefly present some background information about Petri nets. We also describe one type of extended Petri net that we use in our later models. The definitions and theorems are derived from some standard Petri net literature [4,5].

Definition 1: A *Petri net* is a bipartite directed graph that can be represented as a 5-tuple PN = (P, T, F, W, M_0), where:

- P = {p_1, p_2,...,p_m} is a finite set of nodes that represents Petri net places. Place nodes model various conditions or states that relate to the level of abstraction of interest for the system being considered. A place can contain markers, called tokens. When a place does contain tokens, we say the place is marked. An unmarked place contains zero tokens.

$T = \{t_1, t_2,...,t_n\}$ is a finite set of nodes that represent transitions. Transitions model various events that relate to the system being modeled.

$F \subseteq (PxT) \cup (TxP)$ is a flow control function which represents the edges in the graph. The directed edges in F connect places to transitions and transitions to places (but never places to places or transitions to transitions).

W : F -> {1, 2, 3,...} is the weight function with W(a)=0 for $a \notin F$, which specifies the number of tokens that move through the directed edge (explained below).

M_0 : P -> {0, 1, 2,...} is the initial marking, or initial distribution of tokens to places, of the graph. This represents the initial state of the system modeled by the Petri net graph.

The dynamic evolution of a marked Petri net is dictated by the firing of transitions. A transition *t* can fire only when it is enabled, which means that each of *t*'s input places has a "sufficient" number of tokens. The sufficient number depends on the weight function for the arc from place *p* to transition *t*. When a transition *t* fires, tokens are removed from all of *t*'s input places and tokens are deposited in each of *t*'s output places. The number of tokens that are removed and deposited is based on the weight function associated with the edge connecting the transition *t* and the input and output places. The graphical convention for Petri net models calls for the use of circles to represent places, bars for transitions, and dots for tokens. All the arcs are directed showing their source and destination (place to transition or transition to place).

Definition 2: The set of *reachable markings* from M_0 in a Petri net (N, M_0) is denoted by R(N, M_0) or simply R(M_0).

Definition 3: An n-vector of positive integers is defined as a *firing count vector*, whose *i*th element denotes the number of firings of the *i*th transition.

Definition 4: For a Petri net with *n* transitions and *m* places, the *incidence matrix* A = $[a_{ij}]$ is an *n* x *m* matrix of integers and its typical entry is given by

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-}$$

where $a_{ij}^{+} = w(i,j)$ is the weight of the arc from transition *i* to its output place *j* and $a_{ij}^{-} = w(j,i)$ is the weight of the arc to the transition *i* from its input place *j*.

Definition 5: An m-vector y, (n-vector, x) of integers is called an *S-invariant* (*T-invariant*) if Ay = 0 ($A^T x = 0$).

Intuitively, the invariants mean the following. The i-th entry of the S-invariant *y* is a weight y(i) associated with the i-th place such that the weighted sum of tokens remains the same for all the markings reachable from an initial marking. Given a T-invariant *x*, there exist a marking M_0 and a transition firing sequence, such that the i-th entry of *x* is the number of times that the i-th transition fires in the firing sequence.

Theorem 1: An m-vector *y* is an S-invariant iff $M^T y = M_0^T y$ for any fixed initial marking M_0 and any reachable marking M in R(M_0).

Theorem 2: An n-vector *x* is a T-invariant iff there exist a marking M_0 and a firing sequence σ from M_0 back to M_0, and *x* defines the firing count vector for σ.

Definition 6: The set of places (transitions) corresponding to non-zero entries in an S-invariant $y \geq 0$ (T-invariant $x \geq 0$) is called the *support of an invariant* and is denoted by ||y|| (||x||).

Definition 7: An invariant is said to be *minimal* if no proper non-empty subset of the support is also a support. Given a minimal support of an invariant, there is a unique minimal invariant corresponding to the minimal support. Such an invariant is called the *minimal-support invariant.*

Theorem 3: The upper bound on the number of tokens that can ever exist in a place *p* is Min [$M_0{}^T y_i$ / $y_i(p)$], where the minimum is taken over all non-negative minimal-support S-invariants y_i, and yi(*p*) denotes the weight associated with place *p* in S-invariant yi.

Timed Petri nets are Petri nets with the additional feature of time associated with the transitions. There are many variations of Timed Petri nets [4, 7, 14, 15]. We are using Merlin's timed Petri nets [15] because there is a natural mapping for the definition of Merlin's timed Petri nets to the timing features of Ada semantics that we will consider. A timed Petri net (based on Merlin's definition) is a Petri net where each transition *t* is associated with a time interval $[t_{min}, t_{max}]$. Assuming that transition *t* becomes enabled at time t_0 then its firing is allowed to take place only sometime between (t_0+t_{min}) and (t_0+t_{max}). In other words, an enabled transition cannot fire until a minimal time, t_{min}, has elapsed, and also it cannot fire after a maximal time, t_{max}. So the enabled transition can fire only between the time interval t_{min} and t_{max}.

Untimed Petri nets can be viewed as timed nets where the time intervals associated with all the transitions are t_{min} = 0 and t_{max} = infinity. In this paper we consider all sink transitions, which are transitions that have no output places, to have the associated timing tuple $[t_{min} = 0, t_{max} = 0]$. This ensures that a sink transition fires as soon as it becomes enabled -- this is used in some of the net models to discard “useless” tokens. For all other transitions in our models, if the transition does not have a timing tuple explicitly specified, the timing tuple is assumed to be $[t_{min} = 0, t_{max} = T]$ by default, where T is some arbitrary maximum time for all basic operations.

Another form of extension to Petri nets is the use of inhibitor arcs [4]. An inhibitor arc connects a place to a transition and defines the property that the transition associated with the inhibitor arc is enabled only when there are no tokens in the input place. This kind of place testing is known as a "zero testing capability." No tokens are moved through an inhibitor arc when a transition fires. By standard convention, an inhibitor arc is graphically represented by the use of a small circle in place of the arrow head pointing to a transition node. In this paper, we will represent an inhibitor arc as a dotted arc (to make them show up more clearly) with a small circle. Use of the inhibitor arcs allows us to produce less complicated models than would otherwise be possible.

3. Ada Tasking Models

3.1 Entry Call Model for Advanced Tasking Constructs

In Ada, the program units used to support concurrent programming are called tasks. A task can act as a client or a server, or even both of them. As stated in Section 2.1, communications between tasks can be accomplished through two mechanisms, which are rendezvous and protected object. Using rendezvous, two tasks can communicate with each other directly by entry calls, while using protected object, two tasks can only communicate with each other indirectly through a protected object as shared data. The protected object construct is a new feature added to Ada-95, and it operates as a server. In addition to providing remote procedures and functions for clients, a protected object can also provide entries, which means a task can make an entry call to a protected object. This mechanism makes it possible for two tasks to communicate with each other indirectly by both having rendezvous with the same protected object at different times.

A regular entry call can be modeled by the Petri net shown in Figure 1 (a). A task (client) making an entry call blocks and waits for the called task (server) to accept and finish servicing the call. After the rendezvous, both tasks can continue their executions in parallel.

Although this regular entry call model properly captures the behavior of a client-server interaction, it is not sufficient for the advanced tasking constructs of interest in this paper.[2] For instance, a task using the asynchronous transfer of control construct needs to know if the rendezvous can be started immediately. This status checking for rendezvous is not necessary in our regular entry call model. To make this checking possible, we evolve our entry call model by adding a checking step before initiating the rendezvous. The new entry call model is shown in Figure 1 (b). In this new model, the client first sends an entry call request to the server; this action is represented by putting a token in place *entry_call_request*. If the rendezvous is available immediately (i.e., there is a token in place *start_server*), the transition *accept_entry_call* is enabled. The firing of this transition puts a token in place *accept.* Now, both places *entry_call* and *accept* have a token, so the transition *start_rendezvous1* is enabled. When the transition *start_rendezvous1* fires, it puts a token in place *entry_call_ready,* and initializes the rendezvous on the server side. After the accepted entry call finishes its execution, the transition *end_rendezvous2* fires and deposits a token into place *entry_call_return*. Then the transition *end_rendezvous1* is enabled, and it's firing puts a token in place entry*_call_complete*. Finally, the transition *end_entry_call* fires and finishes the whole entry call process.

By using this new entry call model, modeling advanced tasking constructs, such as asynchronous transfer of control, becomes possible. Furthermore, this form of model provides for a consistent interface between various client and server instances. This consistency of the entry call interface make it possible for us to design compositional models, and it will be seen in the following sections.

---

[2] For simplicity, we assume interactions involving only one client at a time; multiple calls can be modeled by using colored Petri nets like those used for the protected object model in Section 3.2.2.

Usually, a client becomes queued when it is waiting for service at a particular entry. However the queuing order, such as FIFO or priority queuing, depends on the Ada implementation and whether or not the Ada-95 real-time annex is being used [17]. To create implementation independent models, and to preserve simple interfaces needed to ensure compositionality, we do not explicitly model entry queues. This means that our models assume that any potential entry call could end up as the first call in the queue. This is quite reasonable for behavior modeling since factors that would determine the queuing order are nondeterministic at this stage of interest.

3.2 Advanced Tasking Constructs and Models

We can now discuss the various models that we developed for the tasking constructs in Ada-95 that are extensions to the Ada-83 standard. These changes in the Ada definition are elaborated in [16,17]. All of the models developed here are subjected to some correctness evaluation using Petri net invariant calculations. We will show how these evaluations help ensure confidence that the models correctly capture the intended behaviors.

3.2.1 Asynchronous Transfer of Control

The asynchronous transfer of control (ATC) allows an activity to be abandoned when a condition arises and an alternative path is to be taken. The syntax is as follows:

```
select
      triggering_alternative;
then abort
      abortable_part;
end select;
```

The triggering alternative can be an entry call statement followed by an optional sequence of statements, or a delay statement followed by an optional sequence of statements. The abortable part is an arbitrary sequence of statements.

We first discuss the case when the triggering alternative is a delay statement. Consider the following example:

```
...
select
      delay d;
      Put_Line (“Delay expired”);         --seq1
then abort
      Calculation;                        --seq2
end select;
Put_Line(“End of ATC with Delay”);        --seq3
...
```

When the program hits the select statement the delay and the calculation are started. Whenever one completes, the other is aborted. This behavior is modeled in Figure 2. Since the abortable part (seq2) may contain any statements except an accept statement [17], the abortable part may not be able to be aborted immediately. For instance, when the abortable part is executing an entry call, the abortion must wait until the abortable part finishes its execution of the entry call. For this reason, we divide the code segment seq2 into *n* atomic sub-sequences seq2_i, where $1 \le i \le n$. Each atomic sub-sequence seq2_i is indivisible, therefore it cannot be aborted when it is executing. The abortion succeeds only when the currently running sub-sequence seq2_i finishes its execution.

Now, let's return to our net model. When the asynchronous select statement is reached, a token is put in *ATC_select* place. The transition *start_delay_and_abortable_part* fires and puts tokens in places *delay_in_progress* and *seq2_1*. The transition *delay_expires* is enabled but it can fire only when the delay time, *d*, expires. When the transition *delay_expires* fires, it starts the execution of code segment seq1, which follows the delay statement immediately, and sends a signal "abandon seq2" to the abortable part. This signal places a token in the place *seq2_abandon*. Now, the transition *abandon_seq2* is enabled. When it fires, it puts a token into each place *seq2_i_abandon* associated with each atomic sub-sequence seq2_i, where $1 \le i \le n$. If seq2_i is currently being executed, i.e., there is a token in place *seq2_i*, both transitions *ignore_abandon* and *abandon_seq2_i* are disabled. When seq2_i finishes its execution, the transition *end_seq2_i* fires and puts a token in place *seq2_i_complete*. Then the transition *abandon_seq2_i* becomes enabled; in the mean time, the transition *start_next* has been disabled because there has already been a token in place *seq2_i_abandon*. The immediate firing of transition *abandon_seq2_i* indicates that seq2_i is abandoned, which then abandons the whole abortable part (seq2). Note that for those sub-sequences seq2_j that are not currently executing, the transition *ignore_abandon* is enabled because neither place *seq2_j* nor place *seq2_j_complete* has a token, but there is a token in place *seq2_j_abandon*. In this case, the token in place *seq2_j_abandon* is discarded immediately by firing the transition *ignore_abandon*.

In a different situation, the execution of code segment seq2 may finish before the delay expires, in which case the abortable part sends a signal "complete seq2" to the triggering part and puts a token in place *seq2_complete*. This enables the transition *seq2_complete_before_delay*, and its firing removes the token from place *delay_in_progress*. In this case, the transition *delay_expires* is disabled and the code segment seq1 can never be executed. This models the behavior of the delay statement that is abandoned when the abortable part finishes before the delay expires. In the special case that the delay expires at the same time that the abortable part finishes, the signal "abandon seq2" is still sent to the abortable part and is ignored by all sub-sequences seq2_i, where $1 \le i \le n$. Since there is a token in place *seq1* at this time, the code segment seq1 is being executed. In either case, one of the transitions, *seq2_complete_before_delay* or *end_seq1*, fires. A token is put in place *ATC_complete* , which enables the transition *end_ATC_select* . This transition fires and code segment seq3 will be executed next. We see that the semantics of the ATC with a triggering statement that is a delay statement is properly modeled.

If the triggering statement is an entry call, then the abortable part has to be aborted if the entry call is accepted and returned. The ATC in this case starts with the issue of an entry call, and the entry call is evaluated first without being

executed. If the call is accepted immediately then the abortable part is never started. But if the call can not be made immediately, either because the server has not reached the accept statement or it is currently servicing the entry call for another client, then the abortable part is executed. If the abortable part completes before the completion of the entry call, an attempt is made to cancel the call. If the attempt to cancel the call succeeds, the ATC is complete. If the abortable part is started but not yet completed and the entry call completes other than due to cancellation, then the abortable part is aborted. If the entry call completes normally, then the optional sequence of statements of the triggering alternative must be executed.

Now consider the following example involving ATC with entry call (as a part of the triggering alternative). The task Client initiates an ATC, issuing an entry call ATC_Event which if accepted by the task Server would result in execution of the Remote_Control at the Server end. If the Server accepts the entry call immediately, then the task Client does not even begin the Normal_Operation (seq2). If the entry call is not accepted immediately, then the Client starts execution of the Normal_Operation. If the execution of Remote_Control finishes before the Normal_Operation, then the Normal_Operation is aborted; if not, an attempt is made to cancel the entry call ATC_Event.

```
task body Server is
begin
      ...
      accept ATC_Event do
            Remote_Control;
      end ATC_Event;
      ...
end Server;

task body Client is
begin
      ...
      select
            Server.ATC_Event;
            Put_Line(“Remote Control done”);          --seq1
      then abort
            Normal_Operation;                         --seq2
      end select;
      Put_Line(“End of ATC with Entry call”);         --seq3
      ...
end Client;
```

The behavior of the above example can be interpreted in terms of the following four cases:

Case 1: the rendezvous is available immediately:
Server.ATC_Event is issued,
Remote_Control is executed,
Put_Line("Remote Control done") is executed, --seq1
Put_Line("End of ATC with entry call") is executed. --seq3

Case 2: no rendezvous starts before seq2 finishes:
Server.ATC_Event is issued,
Normal_Operation is executed, --seq2
Server.ATC_Event is canceled,
Put_Line("End of ATC with entry call") is executed. --seq3

Case 3: the rendezvous finishes before seq2 finishes:
Server.ATC_Event is issued,
partial execution of Normal_Operation occurs concurrently with Remote_Control,
Normal_Operation is aborted and finalized,
Put_Line("Remote Control done") is executed, --seq1
Put_Line("End of ATC with entry call") is executed. --seq3

Case 4: the rendezvous finishes after seq2 finishes:
Server.ATC_Event is issued,
Normal_Operation is executed concurrently with partial execution of Remote_Control,
Server.ATC_Event cancellation is attempted and fails,
Remote_Control execution completes,
Put_Line("Remote Control done") is executed, --seq1
Put_Line("End of ATC with entry call") is executed. --seq3

The model for the above example is shown in Figure 3, in which the modeling of the abortable part is exactly the same as in Figure 2 and the modeling of task Server is ignored because it is the same as in Figure 1 (b). When the asynchronous select statement is reached, a token is put in *ATC_select* place. The transition *make_entry_call* fires and puts a token in both places *abortable_part_ready*and *entry_call_ready*; meanwhile it sends a signal to the server to see if the rendezvous is available immediately. If the rendezvous is available immediately, then a token will be put back in *accept* place, enabling the transition *start_rendezvous*, which when fires starts the rendezvous, modeling the execution of Remote_Control. When the transition *start_rendezvous* fires, the token in place *entry_call_ready* is removed, which enables the transition *ignore_abortable_part* When the transition *ignore_abortable_part* fires, the token in place *abortable_part_ready*is removed immediately, and the abortable part can never be started. After the entry call returns, the transition *end_rendezvous* fires, and the execution of code segment seq1 starts; meanwhile a signal "abandon seq2" is sent to the abortable part. This signal is ignored by the abortable part because at this time, there are no tokens in any of the places *seq2_i* or *seq2_i_complete*, where $1 \le i \le n$. Finally, the transition *end_seq1*

fires, puts a token in place *ATC_complete*, and enables the transition *end_ATC_select*. The transition *end_ATC_select* fires and the code segment seq3 is executed next. This sequence of firings models *Case 1* of the program interpretation.

Note that we have associated a time delay d with the transition *start_abortable_part* for our modeling convenience. If during the time delay d, the transition *start_rendezvous* does not fire, therefore the token in place *entry_call_ready* is not removed, we assume that the rendezvous is not available immediately. In this case, the transition *start_abortable_part* fires and the code segment seq2 is executed; on the other hand, if the transition *start_rendezvous* fires during the time delay d and the token in place *entry_call_ready* is removed, the token in place *abortable_part_ready* can be removed immediately by firing the enabled transition *ignore_abortable_part* Thus, the transition *start_abortable_part* is disabled, and can never be fired, as in *Case 1.*

If the rendezvous is not readily available (i.e., the transition *start_rendezvous* does not fire during the time delay d), the transition *start_abortable_part* fires. The firing of this transition puts a token in place *seq2_1*, modeling the execution of Normal_Operation (seq2). When the Normal_Operation finishes its execution, a signal of "complete seq2" is sent from the abortable part and a token is put in place *seq2_complete*. At this time, if the entry call is still not accepted, the transition *complete_before_rendezvous* is enabled and ready to fire. Its firing takes away the token in *entry_call_ready,* which models successful cancellation of the entry call, also it puts a token in place *ATC_complete*. At some time later, a token will be put back in place *accept* by the task Server when the rendezvous becomes available. However at that time, there is no token in place *entry_call_ready,* so the transition *ignore_accept* is enabled. When it fires, the transition in place *accept* is discarded immediately. Finally the transition *end_ATC_select* fires and the code segment seq3 is executed next. This sequence of firings corresponds to *Case 2* of the program interpretation.

Now consider the situation where the rendezvous completes successfully but the Normal_Operation is still executing. This is modeled when the transition *end_rendezvous* fires, there is a token still in one of the places *seq2_i* or *seq2_i_complete*, where $1 \leq i \leq n$. In this case, the execution of code segment seq1 starts and a signal "abandon seq2" is sent to the abortable part at the same time. In the abortable part, let seq2_i be the atomic sub-sequence in seq2 which is currently being executed (i.e., there is a token in place *seq2_i*), then after the firing of the transition *end_seq2_i*, the token in place *seq2_i* is moved to place *seq2_i_complete*. This enables the transition *abandon_seq2_i*, and both tokens in place *seq2_i_complete* and *seq2_i_abandon* are removed immediately when it fires. This models the successful cancellation of the abortable part. Finally, the execution of code segment seq1 completes, and the code segment seq3 is executed next. This sequence of firings defines *Case 3* of the program interpretation.

The last sequence of firings will model *Case 4* of the program interpretation. If the execution of code segment seq2 completes but the call has not returned yet, then an attempt to cancel the call is made. The attempt to cancel the call is modeled as the possibility of firing *complete_before_rendezvous*. But this transition cannot fire because the call has been already accepted and Remote_Control is being executed, therefore there is no token in place *entry_call_ready.* At the same time, the transition *complete_during_rendezvous* is enabled, because there is no token in place

entry_call_ready and there is a token in place *seq2_complete*. The immediate firing of this transition models the completion of the execution of Normal_Operation. Then, later when the entry call returns, code segment seq1 will be executed and completes. Finally, the transition *end_ATC_select* fires and the code segment seq3 is executed next.

As compared to the model of ATC with delay, this model is more complex. The reason is that in ATC with entry call, the number of alternative behaviors provides more unique execution paths than can be encountered by any execution of an ATC with delay statement.

We now apply some Petri net analysis techniques to the model to establish some correctness properties. To support the evaluation, we consider a closed system involving the ATC. By closed system we mean that the model has an iteration characteristic, i.e., rather than executing the next statement sequence following the ATC select statement, control is returned to the beginning of the ATC select statement by looping back to the initial state.

First, consider the ATC with delay statement model. Assume we have only one atomic sub-sequence seq2_1 in code segment seq2. The model in Figure 2 is now extended to that shown in Figure 4 (a closed system). Since the places being tested by inhibitor arcs are bounded, we can transform the inhibitor arc net into a net without inhibitor arcs using the standard complementary place technique [1]. The initial marking M_0 is [0 1 0 1 1 0 0 0 1 1 0 0 0] and the incidence matrix of this net is as follows:

	a	b	c	d	e	f	g	h	i	j	k	l	m
t1	-1	0	1	0	-1	0	0	0	0	0	0	0	0
t2	0	0	-1	0	1	0	0	0	0	0	0	0	0
t3	0	0	-1	1	1	0	-1	0	0	0	0	0	0
t4	0	1	0	-1	0	-1	1	0	0	0	0	0	0
t5	0	0	0	1	0	0	-1	1	0	0	0	0	0
t6	0	0	0	0	0	0	0	-1	0	1	1	-1	0
t7	0	-1	0	0	0	1	0	0	-1	-1	0	1	0
t8	0	0	0	0	0	0	0	-1	0	0	0	0	0
t9	0	0	0	0	0	0	0	0	1	0	-1	0	0
t10	1	0	0	0	0	0	0	0	0	1	0	-1	1
t11	0	0	0	0	0	0	0	0	0	0	1	0	-1

Using the definitions and theorems from Section 2.2, we can find that the minimal-support S-invariants are:

	a	b	c	d	e	f	g	h	i	j	k	l	m
y1 = [	0	1	0	0	0	1	0	0	0	0	0	0	0]
y2 = [	0	0	0	1	0	0	1	0	0	0	0	0	0]
y3 = [	0	0	1	0	1	0	0	0	0	0	0	0	0]

y4 = [0 0 0 0 0 0 0 0 0 1 0 1 0]
y5 = [0 0 0 0 0 0 0 0 1 0 1 1 1]

and the minimal-support T-invariants are:

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11
x1 = [	1	0	1	1	0	0	1	0	1	1	1]
x2 = [	0	0	0	1	1	1	1	0	1	0	0]
x3 = [	1	1	0	1	1	0	1	1	1	1	1]

From the definition of an S-invariant, given in Section 2.2, the invariant y1 can be interpreted to indicate that in any reachable state from M_0, the weighted sum of tokens in places *b* and *f* is a constant. Similarly, the invariant y2 indicates that the weighted sum of tokens in places *d* and *g* is a constant, and so on. From the initial marking M_0 = [0 1 0 1 1 0 0 0 1 1 0 0 0], we can directly observe that the weighted sum of all five invariants is 1. Obviously, the S-invariant y1 results from the complementary place *b* introduced in the transformation to remove the inhibitor arc, and whatever transition fires, the weighted sum of tokens in the complementary place *b* and its original place *f* is always equal to 1. Similarly, S-invariant y2, y3 and y4 result from the complementary place *d, e* and *j* respectively. The S-invariant y5 tells us that only one of the places *i, l, m* and *k* can have a token at any time. This is true because only one process can be in the triggering part of ATC_select statement, and the state of this process can be one of the following: *ATC_select, delay_in_process, seq1* and *ATC_complete* . Note that if the delay statement is abandoned due to the completion of code segment seq2, the *ATC_complete* state is reached directly without execution of code segment seq1. There is no corresponding S-invariant for the abortable part. The reason for this is that the abortable part can be abandoned when the code segment seq1 in the triggering part is still running (i.e., ATC has not completed yet), and we may possibly lose tokens in the abortable part due to the cancellation.

The above minimal-support T-invariants cover all the transitions in the net, and it implies that our net model is live. For each minimal-support T-invariant in our example, there exists a corresponding firing sequence by which the initial marking M_0 = [0 1 0 1 1 0 0 0 1 1 0 0 0] can be reproduced. The firing sequences and their physical explanations are as follows:

Firing sequence 1:
<*start_delay_and_abortable_part*(t7), *delay_expires* (t10), *abandon_seq2* (t1), *end_seq2_1* (t4), *abandon_seq2_1* (t3), *end_seq1* (t11), *end_ATC_select* (t9)>
This is the case when delay expires before seq2 finishes, thus the abortable part (seq2) is abandoned.

Firing sequence 2:
<*start_delay-and_abortable_part* (t7), *end_seq2_1* (t4), *start_next* (t5), *seq2_complete_before_delay* (t6), *end_ATC_select* (t9)>

This is the case when the abortable part finishes before the delay expires, thus the delay is abandoned and code segment seq1 is not executed.

Firing sequence 3:
<*start_delay_and_abortable_part* (t7), *end_seq2_1* (t4), *delay_expires* (t10), *start_next* (t5), *ignore_complete* (t8), *abandon_seq2* (t1), *ignore_abandon* (t2), *end_seq1* (t11), *end_ATC_select* (t9)>
This is the special case, where the delay expires at the same time as seq2 finishes. In this case, seq1 is executed, and an "abandon seq2" signal is sent to the abortable part seq2, which is then ignored by the abortable part.

Let us now consider the model of ATC with entry call. We derive the net shown in Figure 5 from that of Figure 3 by closing the system and also eliminating inhibitor arcs. We also assume that there is only one atomic sub-sequence seq2_1 in code segment seq2. The initial marking M_0 is [0 1 0 1 1 0 0 0 0 0 1 1 1 0 1 0 0 0] and the incidence matrix of the net shown in Figure 5 is as follows:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
t1	-1	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
t2	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
t3	0	0	-1	1	1	0	-1	0	0	0	0	0	0	0	0	0	0	0
t4	0	1	0	-1	0	-1	1	0	0	0	0	0	0	0	0	0	0	0
t5	0	0	0	1	0	0	-1	0	1	0	0	0	0	0	0	0	0	0
t6	0	-1	0	0	0	1	0	-1	0	0	0	0	0	0	0	0	0	0
t7	0	0	0	0	0	0	0	0	-1	1	0	1	1	-1	1	0	0	0
t8	0	0	0	0	0	0	0	1	0	0	-1	-1	-1	1	-1	1	0	0
t9	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
t10	0	0	0	0	0	0	0	0	0	-1	1	0	0	0	0	0	0	0
t11	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0
t12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0
t13	0	0	0	0	0	0	0	0	0	0	0	1	1	-1	1	-1	1	0
t14	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	1
t15	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	-1

From the definition in Section 2.2, the following minimal-support S-invariants can be computed:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
y1 = [	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0]
y2 = [	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0]
y3 = [	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0]
y4 = [	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0]
y5 = [	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0]

y6 = [0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
y7 = [0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1]

Again from the definition in Section 2.2, the following minimal-support T-invariants can be computed:

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15
x1 = [	1	1	0	0	0	0	0	1	1	1	0	0	1	1	1]
x2 = [	0	0	0	1	1	1	1	1	0	1	0	1	0	0	0]
x3 = [	1	0	1	1	0	1	0	1	0	1	0	0	1	1	1]
x4 = [	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1]

The S-invariants y1-y6 result from the complementary places *b, d, e, l, m,* and *o* introduced in the transformation to remove inhibitor arcs. For example, y1 tells us that the weighted sum of tokens in places *b* and *f* is a constant. From the initial marking, M_0 = [0 1 0 1 1 0 0 0 0 0 1 1 1 0 1 0 0 0], we can directly see that the weighted sum of tokens for all six S-invariants y1-y6 is 1.

S-invariant y7 signifies that in any state, the weighted sum of tokens in places *k, n, q, r* and *j* is a constant. As noted before, the initial marking let us reason that the weighted sum of tokens in these places must be 1. Thus, there can be a token in only one of these places at any time. This means that there are five states associated with the triggering part and these states are mutually exclusive. The five states are as follows:

1) the ATC_select is ready to be initiated,
2) the task has issued the ATC and the entry call is ready,
3) the task has issued the ATC and the rendezvous has begun to execute,
4) the task has issued the ATC, the rendezvous has finished and seq1 is being executed,
5) the task has issued the ATC and the ATC completes with the rendezvous and seq1 either being executed or not.

Note that state (5) can be reached from state (2) directly if the entry call is canceled. Similarly as before, there is no corresponding S-invariant for the abortable part. The reason is the same, i.e., the abortable part can be abandoned when the code segment seq1 in the triggering part is still running, and we may possibly lose tokens in the abortable part due to the cancellation.

The above minimal-support T-invariants cover all the transitions in the net, and it implies that our net model is live. As before, for each minimal-support T-invariant in our example, we have a corresponding firing sequence, by which the initial marking M_0 = [0 1 0 1 1 0 0 0 0 0 1 1 1 0 1 0 0 0] can be reproduced. The firing sequences and their physical explanations are as follows:

Firing sequence 1:

<*make_entry_call* (t8), *start_rendezvous* (t13), *ignore_abortable_part* (t9), *end_rendezvous* (t14), *abandon_seq2* (t1), *ignore_abandon* (t2), *end_seq1* (t15), *end_ATC_select* (t10)>
This is the case when the rendezvous is available immediately .

Firing sequence 2:
<*make_entry_call* (t8), *start_abortable_part* (t6), *end_seq2_1* (t4), *start_next* (t5), *complete_before_rendezvous* (t7), *ignore_accept* (t12), *end_ATC_select* (t10)>
This is the case when no rendezvous starts before seq2 finishes.

Firing sequence 3:
< *make_entry_call* (t8), *start_abortable_part* (t6), *start_rendezvous* (t13), *end_rendezvous* (t14), *abandon_seq2* (t1), *end_seq2_1* (t4), *abandon_seq2_1* (t3), *end_seq1* (t15), *end_ATC_select* (t10)>
This is the case when the rendezvous finishes before seq2 finishes.

Firing sequence 4:
<*make_entry_call* (t8), *start_abortable_part* (t6), *start_rendezvous* (t13), *end_seq2_1* (t4), *start_next* (t5), *complete_during_rendezvous* (t11), *end_rendezvous* (t14), *abandon_seq2* (t1), *ignore_abandon* (t2), *end_seq1* (t15), *end_ATC_select* (t10)>
This is the case when the rendezvous finishes after seq2 finishes.

The four T-invariants correspond to the four cases that can arise during an ATC with an entry call, and they are the same as those identified earlier and discussed in detail. In the special case that the rendezvous and the execution of seq2 end at the same time, the firing sequence is similar to *Firing sequence 4* and corresponds to the same minimal-support T-invariant x4.

3.2.2 Protected Object

A protected object provides coordinated access to shared data through calls on its visible operations, which can be protected subprograms or protected entries [2]. It gives exclusive read-write or concurrent read-only access to the shared data. If access is requested by more than one task for a protected entry or a procedure, only one of the requesting tasks gets write-only access to the called entry or procedure. But, if the access requested by more than one task is for a protected function, all of the calling tasks can obtain read-only access to the function.

Depending on the type of call, the level of access is determined. The entry calls and procedure calls attempt to gain exclusive read-write access while function calls attempt read-only access. A new protected action cannot be started on a protected object while another protected action is underway, unless both actions are the result of a call on a protected function. Starting a protected action corresponds to acquiring the execution resource associated with the protected object, either for concurrent read-only access, if the protected action is for a call on a protected function, or for

exclusive read-write access, otherwise. Completing the protected action corresponds to releasing the associated execution resource.

Consider an example in which we have a protected object that contains an entry, a procedure and a function.

```
protected x is
      entry entry_x;
      procedure procedure_x;
      function function_x return a_variable;
private
      p,q : integer;
end x;

protected x body is
      entry entry_x when a_barrier is
      begin
      ...
      end entry_x;
      procedure procedure_x is
      begin
      ...
      end procedure_x;
      function function_x return a_variable is
      begin
      ...
      end function_x;
end x;
```

This example can be modeled as shown in Figure 6. The places *write_control* and *read_control* provide appropriate access to the protected entry_x, procedure_x and function_x in protected object x. In the initial marking, *write_control* has a token, but *read_control* does not. Both the transitions *accept_entry_call* and *start_procedure_call* have an inhibitor arc from the *read_control* place. Thus, these transitions are enabled only when there is a token in the *write_control* place and there are no tokens in the *read_control* place. This reflects the semantics that a protected entry or a protected procedure can be accessed only when there is no other protected action being carried out. Ada semantics allow simultaneous access of protected functions. Whenever a function call is made, then there is a token in the place *function_call_ready.* The transition *start_function_call* is enabled only when there are tokens in the *write_control* place and *function_call_ready* place. When transition *start_function_call* fires it takes away the token from the *write_control* place and puts a token in the *read_control* place. Also a token is put back in the *write_control* place, so as to provide simultaneous access to other tasks that request the protected function. So, when the function is being

executed there is a token in *read_control*, thus disabling the firing of *accept_entry_call* and *start_procedure_call* transitions. When the function's execution is completed, the *end_function_call* transition fires. The firing of this transition removes the token from the *read_control* place, thus releasing the resource.

Entry calls are given exclusive read-write access. But, for this to occur, there must be no other protected action in execution. The modeling is done as follows. Whenever there is a protected entry call, there is a token in *entry_call_request* place. The barrier is then evaluated, if the barrier condition is true, a token will be put in place *entry_call_ready*, which means that the server is ready to accept this entry call; otherwise, the barrier condition will be re-evaluated at sometime later, meanwhile the entry call is suspended temporally. To simplify our model, we ignore the modeling of re-evaluation for the barrier and assume that the transition *barrier_evaluation* will not fire until the barrier condition becomes true. When there is a token in place *entry_call_ready*, the transition *accept_entry_call* is enabled as long as there is a token in *write_control* place and no token in *read_control* place. The firing of *accept_entry_call* puts a token in place *accept* of the calling task (not shown in this figure), and the transition *start_rendezvous* of the calling task (not shown in this figure) should be enabled at this time. The firing of transition *start_rendezvous* of the calling task puts a token back in place *rendezvous_ready* of module entry_x. The transition *start_rendezvous* of module entry_x is now enabled and the firing of *start_rendezvous* would correspond to the beginning of the execution of the entry body. After the execution is complete, the transition *end_rendezvous* fires. The firing of this transition puts back a token in the *write_control* place, thus releasing the protected body. Module procedure_x has exactly the same interface to place *read_control* and *write_control* as module entry_x, but the module procedure_x itself is much simpler because no barrier needs to be evaluated and it can be started as long as there is a token in place *write_control* and no token in place *read_control*. Note that the interface for the protected entry is the same as the interface for a general server, as illustrated in Figure 1 (b). Thus, the protected object model is compositional with respect to the various forms of client modules.

To simplify our model, we label our incoming/outgoing arcs of each module by colors. For instance, in the function_x module, arcs to/from clients are labeled by <f>. This means each function call is represented by a unique color *f*, so the result of a particular function call would be returned to the correct place in the calling task by matching to the same color *f*. Similarly, arcs to/from clients in entry_x and procedure_x modules are labeled by <e> and <p>, respectively, for the same reason.

A closed system for this model is shown in Figure 7. In this transformed model, rather than unfold the places and transitions into several copies, we assume that we have only one color for each entry call, procedure call and function call. So, all colored tokens in Figure 6 are represented by colorless tokens in Figure 7. This assumption is appropriate because we are now concerned with the behavior of the protected object x, while the outside world (the calling tasks that are accessing the protected object x) is outside our scope of consideration.

We further assume that a maximum of k function calls is allowed, where k > 1, thus we can remove the inhibitor arcs by using the complementary place technique [1]. The initial marking is M_0 = [0 0 2 0 0 k k 1 3 0 2 0], which means

that currently there are 2 clients trying to make the entry call entry_x, 2 clients trying to make the procedure call procedure_x, and 3 clients trying to make the function call function_x.

The incidence matrix for the net shown in Figure 7 is as follows:

	a	b	c	d	e	f	g	h	i	j	k	l
t1	-1	1	0	0	0	0	0	0	0	0	0	0
t2	0	-1	1	0	0	0	0	1	0	0	0	0
t3	0	0	-1	1	0	0	0	0	0	0	0	0
t4	1	0	0	-1	0	0	0	-1	0	0	0	0
t5	0	0	0	0	1	-1	-1	0	-1	1	0	0
t6	0	0	0	0	-1	1	1	0	1	-1	0	0
t7	0	0	0	0	0	0	0	-1	0	0	-1	1
t8	0	0	0	0	0	0	0	1	0	0	1	-1

The resulting minimal-support S-invariants are

	a	b	c	d	e	f	g	h	i	j	k	l
y1 = [	0	0	0	0	1	1	0	0	0	0	0	0]
y2 = [	0	0	0	0	1	0	1	0	0	0	0	0]
y3 = [	1	1	1	1	0	0	0	0	0	0	0	0]
y4 = [	0	0	0	0	0	0	0	0	1	1	0	0]
y5 = [	0	0	0	0	0	0	0	0	0	0	1	1]
y6 = [	1	1	0	0	0	0	0	1	0	0	0	1]
y7 = [	0	0	0	0	0	1	0	0	0	1	0	0]
y8 = [	0	0	0	0	0	0	1	0	0	1	0	0]

and the minimal-support T-invariants are

	t1	t2	t3	t4	t5	t6	t7	t8
x1 = [	1	1	1	1	0	0	0	0]
x1 = [	0	0	0	0	1	1	0	0]
x1 = [	0	0	0	0	0	0	1	1]

The S-invariant y1 and y2 result from the complementary places *f* and *g* introduced by the transformation to remove the inhibitor arcs. S-invariant y3 indicates that the weighted sum of tokens in places a, *b*, *c* and *d* is a constant, which can be computed by $y3^{T}M_0 = 2$. This means that each of the two entry calls can only be in one of the following four states: *rendezvous_ready*, *entry_call_in_execution*, *entry_call_request* and *entry_call_ready.* Similarly, S-invariant y4 indicates that the weighted sum of tokens in places *i* and *j* is a constant, which can be computed by $y4^{T}M_0 = 3$, and it

means that each of the three function calls can be in one of two states: *function call_ready* and *function_call_in_execution*, and S-invariant y5 indicates that the weighted sum of tokens in places *k* and *l* is a constant, which can be computed by $y5^T M_0 = 2$, and it means that each of the two procedure calls can be in one of two states: *procedure_call_ready* and *procedure_call_in_execution*. S-invariant y6 indicates that the weighted sum of tokens in places a, *b*, *h* and *l* is a constant, which can be computed by $y6^T M_0 = 1$. This means that at any time, there can be at most one entry call or procedure call being executed. S-invariant y7 indicates that the weighted sum of tokens in places *f* and *j* is a constant, which can be computed by $y7^T M_0 = k$. This means that the number of function calls in execution can not exceed k, otherwise the number of tokens in place *f* will be negative, which is incorrect. S-invariant y8 has exactly the same interpretation as S-invariant y7.

Applying Theorem 3, as stated in Section 2.2, we can find the upper bound of place *b*:

$$UB(b) = \text{Min} [M_0^T y_i / y_i(p)] = \text{Min} [k/0, k/0, 2/1, 3/0, 2/0, 1/1, k/0, k/0] = 1/1 = 1.$$

Similarly, the upper bound of place *j* and place *l* can be computed as follows:

$$UB(j) = \text{Min} [M_0^T y_i / y_i(p)] = \text{Min} [k/0, k/0, 2/0, 3/1, 2/0, 1/0, k/1, k/1] = \text{Min} [k, 3].$$

$$UB(l) = \text{Min} [M_0^T y_i / y_i(p)] = \text{Min} [k/0, k/0, 2/0, 3/0, 2/1, 1/1, k/0, k/0] = 1/1 = 1.$$

Thus, at most one token can be in place *b* and *l*. This conforms to the desired Ada semantics of allowing only one task to execute a procedure body or an entry body in a protected object. Similarly, Min [k, 3] tokens can be in place *j*, which conforms to the Ada semantics of allowing more than one task to execute a function body in a protected object, recalling that k > 1.

The interpretation for the T-invariants are pretty straightforward; here we list the corresponding firing sequence for T-invariants x1, x2 and x3 as follows:

Firing sequence 1:
<*barrier_evaluation* (t3), *accept_entry_call* (t4), *start_rendezvous* (t1), *end_rendezvous* (t2)>

Firing sequence 2:
<*start_function_call* (t5), *end_function_call* (t6)>

Firing sequence 3:
<*start_procedure_call* (t7), *end_procedure_call* (t8)>

These three firing sequences correspond to the execution of an entry call, function call, and procedure call. Each of them will reproduce the initial marking M_0 = [0 0 2 0 0 k k 1 3 0 2 0]. Also, since all the transitions are covered by the minimal-support T-invariants, our transformed net model for the protected object is live.

3.2.3 Requeue Statement

A requeue statement is used to complete an accept statement or entry body, while redirecting the corresponding entry call to a new entry queue. The general syntax for the requeue statement is as follows:

```
requeue entry_name [with abort];
```

We simplify the discussion by considering only the basic form of requeue -- without the optional abort clause. We use the following example to describe our modeling of the requeue statement. The purpose of the example is to demonstrate the use of the requeuing technique and to show how clients and servers could be connected at the Petri net model level.

```
Task body Printer is
...
accept Print_Request do
begin
      if (Print_Type = Color) then
            requeue Color_Printer.Color_Print_Request;
      else
            Print_Document;                          --seq1
      end if;
end Print_Request;
Put_Line(“Ready for next print job”);                --seq2
...
end Printer;

Task body Color_Printer is
...
accept Color_Print_Request do
      Print_Color_Document;                          --seq3
end Color_Print_Request;
Put_Line(“Ready for next color print job”);          --seq4
...
end Color_Printer;

Task body User is
...
Printer.Print_Request;
Put_Line(“Print done ready for next job”);           --seq5
...
end User;
```

In this example, the task User provides some print job to the task Printer, by making an entry call Print_Request. The task Printer accepts the entry call Print_Request and checks if the prints are color. If so, it requeues the entry call to the entry, Color_Print_Request, in the task Color_Printer. After requeuing the entry call, Print_Request, the task Printer executes the statement Put_Line("Ready for next print job"). The task Color_Printer which accepts the entry call Color_Print_Request replies to the task User after processing the entry call Color_Print_Request. If the prints are not color then the task Printer prints the document and replies to the task User. Depending on the condition whether the prints are color or not, the task that replies to the task User is determined. The reason we have chosen this example is that it provides us the opportunity to discuss all the possible scenarios involving the requeue statement.

The above example can be modeled as shown in Figure 8. When the Printer task accepts the entry call made by the User task there is a token in place *condition*. Now, one of the transitions *start_seq1* or *make_requeued_entry_call* fires, depending on the condition of whether or not the Print_Type is color. Assuming that the condition is true, the transition *make_requeued_entry_call* fires and a token is put in both *requeued_entry_call* and *requeued_entry_call_request* places. When the requeued entry is accepted (recall that we have ignored all entry call queues, so the original semantic of "requeue" is simplified as that the rendezvous on server Color_Printer is available immediately), the transition *transfer_entry_call_control* fires, which denotes that the task Printer hands over the sever role for this entry call to the task Color_Printer. A token is put in each of the places *if_statement_complete* and *requeued_entry_call_ready*, and the transition *start_rendezvous3* is enabled and can fire. This denotes that the entry is successfully "requeued". Now, the execution of the entry call Color_Print_Request and the task Printer can proceed in parallel and independent of each other. Since there is a token in place *if_statement_complete*, the transition *end_if_statement* is enabled. So it fires and the statements following the if statement (i.e., code segment seq2) is executed next.

Now consider the behavior associated with the requeued entry. After the requeued entry call has been serviced by task Color_Printer, the transition *end_rendezvous3* is enabled. Firing of *end_rendezvous3* in task Color_Printer puts tokens in places *seq4* (not shown in this figure) and *entry_call_complete* in module task Printer. The task Color_Printer continues to execute code segment seq4. The firing of the enabled transition *return_entry_call* puts a token in place *entry_call_return*. The transition *end_rendezvous1* is now enabled and puts a token in place entry_*call_complete* of module task User when it fires. Finally, the transition *end_entry_call* fires and continues to execute the code segment seq5.

In a different situation, if the condition is false (i.e., the request is not for color printing), the entry call is not requeued and the task Printer has to reply to task User after servicing the call (the standard way of completing an accept of an entry call). The modeling of this case is quite straightforward. The key observation is that the transition *end_rendezvous2* will fire and the entry call is returned by firing the transition *return_entry_call*, similar to the behavior when the entry call was requeued.

As was done in the prior evaluations, we derive Figure 9 from Figure 8 to produce a new model of a closed system. The initial marking M_0 is [1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0], which indicates that all three tasks User, Printer and Color_Printer can be started at the same time.

The incidence matrix of the net shown in Figure 9 is as follows:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
t1	-1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
t2	0	-1	-1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
t3	0	0	0	-1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t4	1	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t5	0	0	1	0	0	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0
t6	0	0	0	0	0	0	0	-1	0	0	1	0	0	0	0	0	0	0	0
t7	0	0	0	0	0	0	0	0	-1	1	0	0	1	0	0	0	0	0	0
t8	0	0	0	1	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0
t9	0	0	0	0	0	0	0	0	0	0	-1	1	0	0	0	0	0	0	0
t10	0	0	0	0	0	0	0	0	1	0	0	-1	0	0	0	0	0	0	0
t11	0	0	0	0	0	0	1	0	0	0	0	0	-1	0	0	0	0	0	0
t12	0	0	0	0	0	0	0	0	0	0	-1	0	0	1	0	1	0	0	0
t13	0	0	0	0	0	0	0	0	0	0	0	0	1	-1	-1	0	0	1	0
t14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-1	-1	0	0
t15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	1
t16	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	-1

The minimal-support S-invariants can be computed as follows:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
y1 = [	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1]
y2 = [	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	1	1]
y3 = [	1	0	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	1	1]
y4 = [	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1]
y5 = [	0	0	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0	0	0]
y6 = [	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1]

and the minimal-support T-invariants are:

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15	t16
x1 = [	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0]

x2 = [1 1 1 1 1 1 0 1 0 0 1 1 1 1 1 1]

S-invariant y1 indicates that the weighted sum of tokens in places *a, f, c, h, k, l, i, p, o, r, s, j, d* and *e* is a constant. From the initial marking M_0 = [1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0], we find that the constant is equal to 1. This shows that M(*l*) + M(*s*) ≤ 1, which means that either seq1 or seq3 can be executed at any time. This is true because, in our example, the entry call made by task User is the only entry call either to task Printer or requeued to task Color_Printer, and it can be executed by either of them but never both. Actually, after task Printer executed the requeue statement, it hands over the control of the entry call to the server Color_Printer, and the code segment seq1 can never be executed. The interpretation of S-invariant y2 is the same, the only difference between S-invariant y1 and y2 is that the place *f* and *c* in the set of places {*a, f, c, h, k, l, i, p, o, r, s, j, d, e*} are replaced by place *b*, and it makes the weighted sum of tokens in places *a, b, h, k, l, i, p, o, r, s, j, d* and *e* is still a constant. This is true in our example because place *b* is redundant, whenever there is a token in place *f* or *c*, there is a token in place b; whenever the token in place *c* is removed, the token in place *b* is also removed. Similarly, S-invariant y3 results from replacing places *p* and *o* with place *n*, and S-invariant y4 results from making both of the replacements as in S-invariant y2 and y3.

S-invariant y5 indicates that the weighted sum of tokens in places *g, c, h, k, l, i, m* and *n* is a constant. This S-invariant can be interpreted that the token in place *g* may go around the direct circuit (*g, c, h, k, l, i, m*) if the requeue statement is not executed, and if the requeue statement is executed, this token will not be lost because the token which is temporally in place *n* will be moved to place *m* after the task Color_Printer takes over the entry call.

S-invariant y6 indicates that the weighted sum of tokens in places *q, o, r* and *s* is a constant. The interpretation is pretty straightforward because it indicates that the token in place *q* can go around the only direct circuit (q, o, r, s) in which place *q* is located.

The T-invariants, x1 and x2, cover all the transitions in the net, and it indicates that our transformed net model is live. The corresponding firing sequences are as follows:

Firing sequence 1:
<*make_entry_call* (t1), *accept_entry_call* (t5), *start_rendezvous1* (t2), *start_rendezvous2* (t6), *start_seq1* (t9), *end_seq1* (t10), *end_rendezvous2* (t7), *end_if_statemnt* (t11), *return_entry_call* (t8), *end_rendezvous1* (t3), *end_entry_call* (t4)>

Firing sequence 2:
<*make_entry_call* (t1), *accept_entry_call* (t5), *start_rendezvous1* (t2), *start_rendezvous2* (t6), *make_rqueued_entry_call* (t12), *accept_requeued_entry_call* (t14), *transfer_entry_call_control* (t13), *end_if_statemnt* (t11), *start_rendezvous3* (t15), *end_rendzvous3* (t16), *return_entry_call* (t8), *end_rendezvous1* (t3), *end_entry_call* (t4)>

T-invariant x1 corresponds to the firing sequence when the call is not requeued, whereas T-invariant x2 corresponds to the firing sequence when the call has been requeued. As with the other models presented, the S-invariants and T-invariants analysis provide confidence in the correctness of the model.

4. Conclusion

In general, formal modeling of advanced tasking and real time constructs in Ada is difficult due to the need to properly capture many behaviors that are interdependent. It is often the case that such models or formal notations are themselves difficult to understand to a reader who is not already very familiar with the specific notation. In this paper, we have presented and discussed models for Ada-95 tasking constructs using the Petri net modeling formalism. Petri nets have been chosen because they tend to provide a simple, easy to understand model, and the underlying mathematics of the model is very powerful for modeling the attributes of concern, most specifically, concurrency, non-determinism, synchronization, and mutual exclusion. We have developed compositional models for the advanced tasking constructs provided by Ada-95, as a complement to existing modeling work for Ada-83. The net models were subjected to some formal evaluation using Petri net invariants. This showed that net-based properties derived directly from the calculation of place and transition invariants correspond to the desired Ada-based behaviors of the tasking primitives.

The basic models provided can be used independently as a way of documenting formal semantics for the various constructs considered, or as building blocks for a net-based analysis of advanced Ada programs. Hopefully, it will be possible to enhance previous net-based analysis techniques and tools developed for Ada-83 to support automated analysis of programs that use the advanced constructs discussed in this paper. This will involve extending current techniques for program-to-net translation, as well as enhancing analysis methods that operate on the produced net models. The net translation is likely to be conceptually straightforward, but the impact of the new constructs on the performance of existing analysis methods (algorithms) is yet to be determined.

References

[1] S. M. Shatz, *Development of Distributed Software: Concepts and Tools,* Macmillan Publishing Company, 1993.

[2] *Ada Reference Manual - Language and Standard Libraries* United States Department of Defense, Ver 6.0, December 1994.

[3] J. Corbett, “Timing Analysis of Ada Tasking Programs,” *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 461-483.

[4] T. Murata, “Petri Nets: Properties, Analysis and Applications,” *Proceedings of the IEEE,* 77(4):541-580, April 1989.

[5] J. L. Peterson, *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, Inc., 1981.

[6] *Ada-83 Language Reference Manual*, United States Department of Defense.

[7] D. Mandrioli, R. Zicari, C. Ghezzi and F. Tisato, "Modeling the Ada Task System by Petri Nets," *Computer Languages*, 10(1):43-61, 1985.

[8] S. M Shatz and W. K. Cheng, "An Approach to Automated Static Analysis of Distributed Software," *Journal of Systems and Software* , 8(5):343-359, December 1988.

[9] S. Duri, U. Buy, R. Devarapalli and S. M. Shatz, "Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada," *ACM Transactions on Software Engineering Methodology* , 3(4):340-380, October 1994.

[10] S. M. Shatz, S. Tu, T. Murata and S. Duri, "An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis," *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1307-1322, December 1996.

[11] A. Wellings and A. Burns, "Implementing Atomic Actions in Ada 95," *IEEE Trans. on Software Engineering*, Vol., 23, No. 2, Feb. 1997, pp. 107-123.

[12] R. K. Gadela and S. M. Shatz, "Modeling of Advanced Tasking in Ada-95: A Petri Net Perspective," *Proc. Of the IEEE 2nd Int. Workshop on Software Eng. For Parallel and Distributed Systems*, May 1997, Boston, Mass., 4-14.

[13] J. Barnes, *Programming in Ada 95*. Addison-Wesley, Inc., 1996.

[14] R. R. Razouk and C. V. Phelps, "Performance Analysis using Timed Petri Nets," *Proceedings 1984 International Conference on Parallel Processing*126-129, August 1984.

[15] P. Merlin, "A Methodology for the Design and Implementation of Communication Protocols," *IEEE Transactions on Communications*, Vol. 24, No. 6, June 1976, pp. 614-621.

[16] *Changes to Ada - 1987 to 1995 : Language and Standard Libraries*, United States Department of Defense, Ver 6.0, December 1994.

[17] A. Burns and A. Wellings, *Concurrency in Ada*, Cambridge Press, 1995.

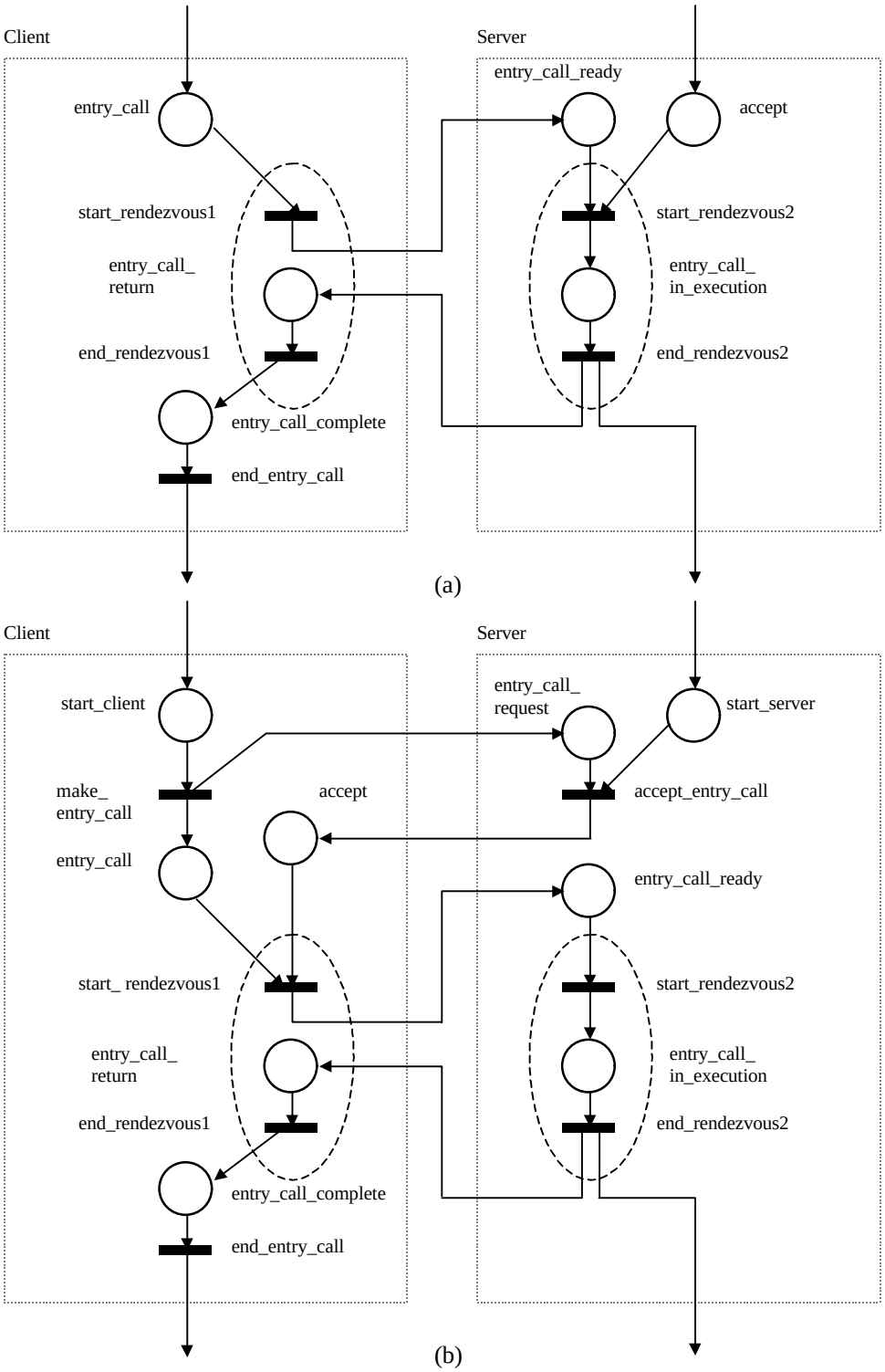


Figure 1. (a) Traditional client-server model for entry call (b) Entry call model for advanced tasking constructs

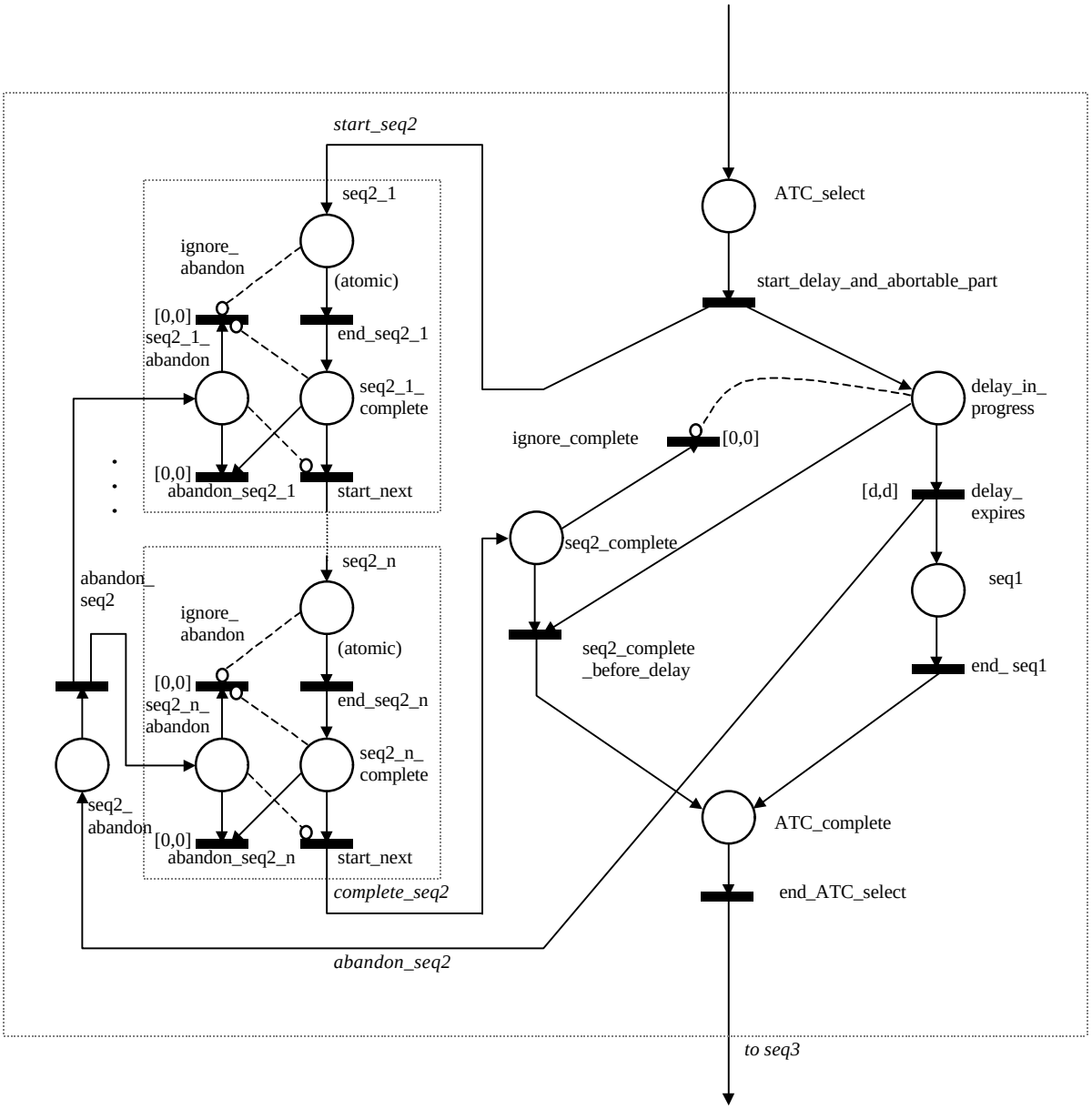


Figure 2 ATC model with delay statement

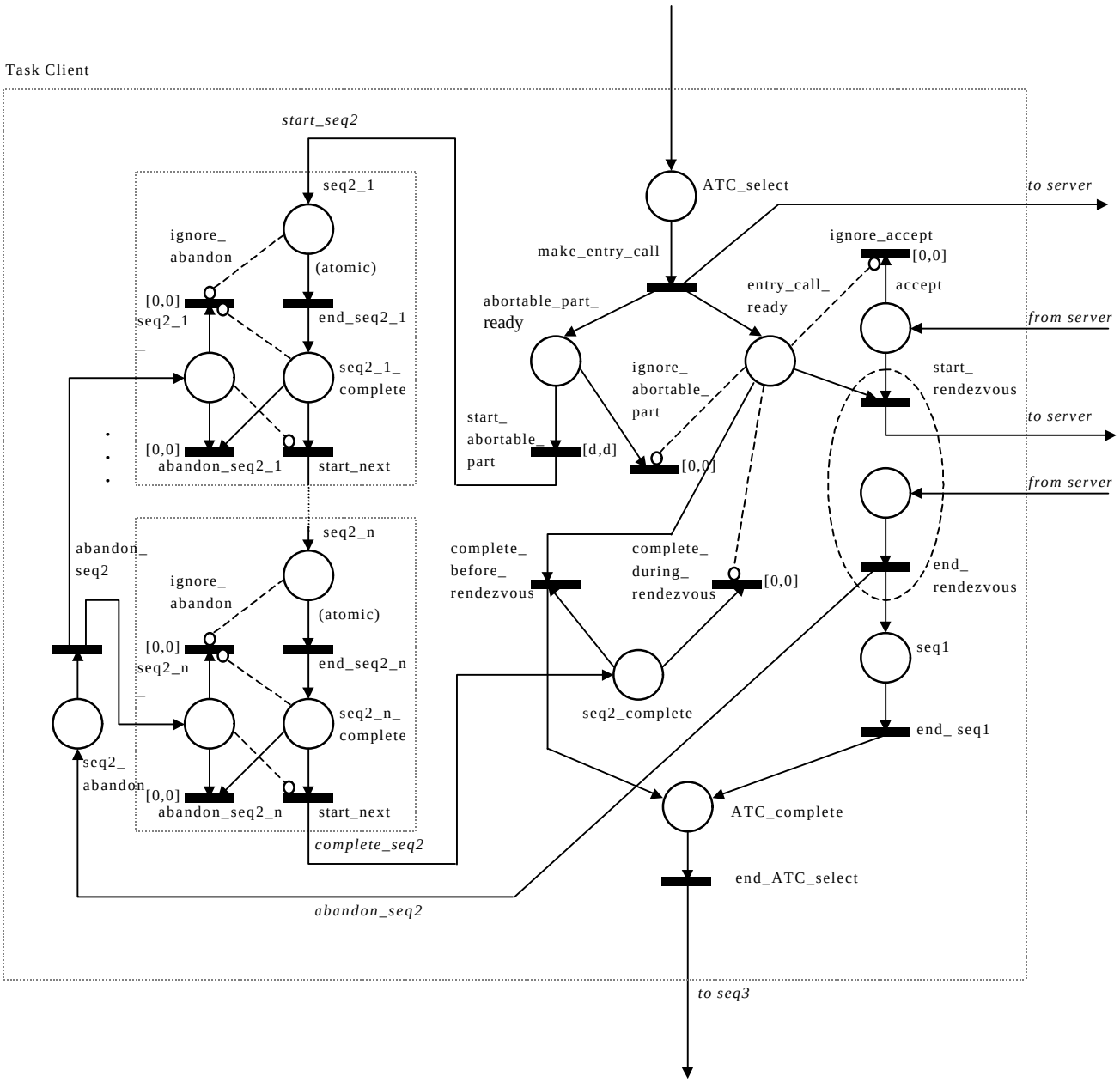
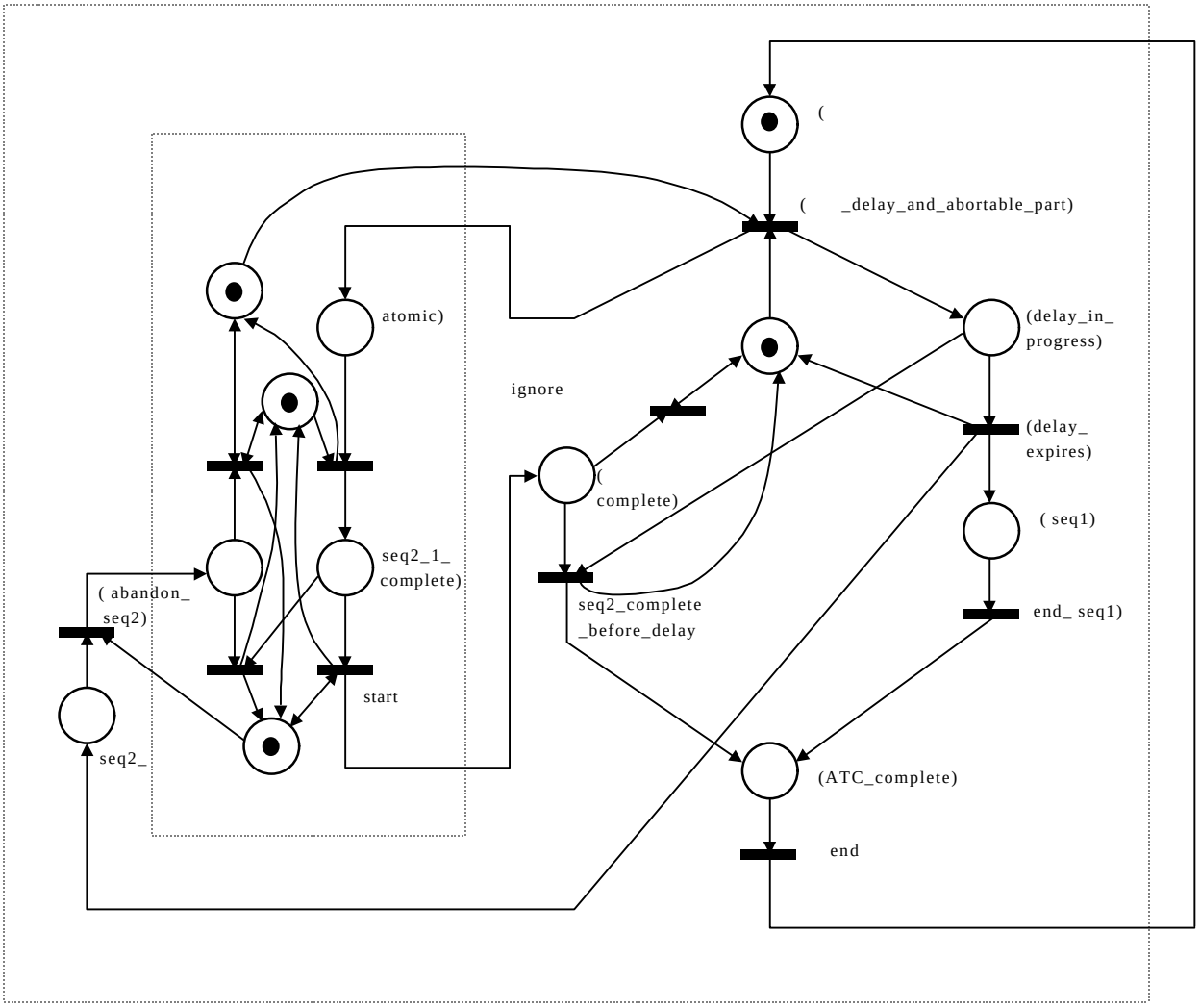


Figure 3 ATC model with entry call


(
(_delay_and_abortable_part)
atomic)
(delay_in_
progress)
ignore
(delay_
expires)
(
complete)
(seq1)
seq2_1_
complete)
(abandon_
seq2)
seq2_complete
_before_delay
end_ seq1)
start
seq2_
(ATC_complete)
end

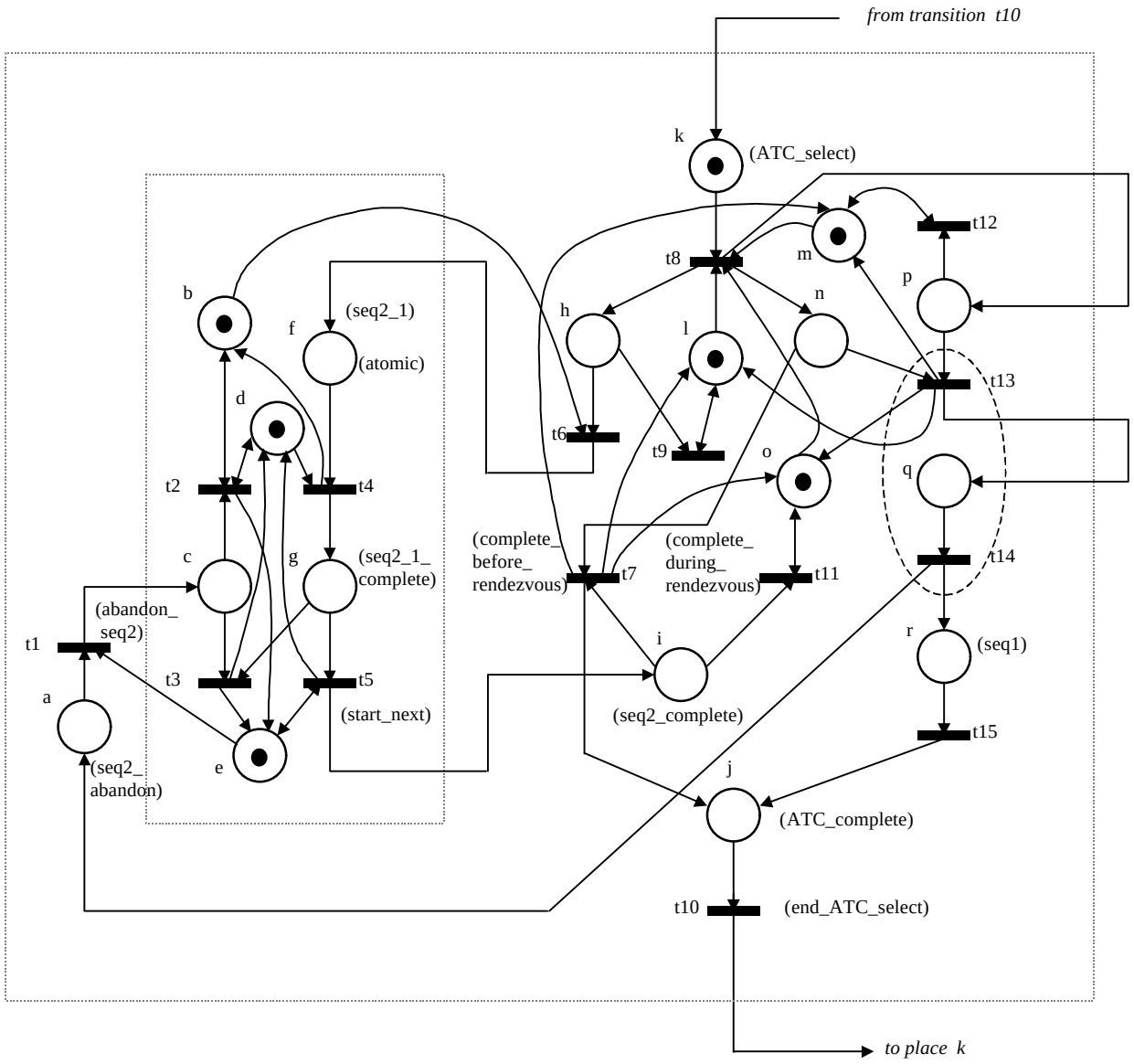


Figure 5 Transformed model of ATC with entry call

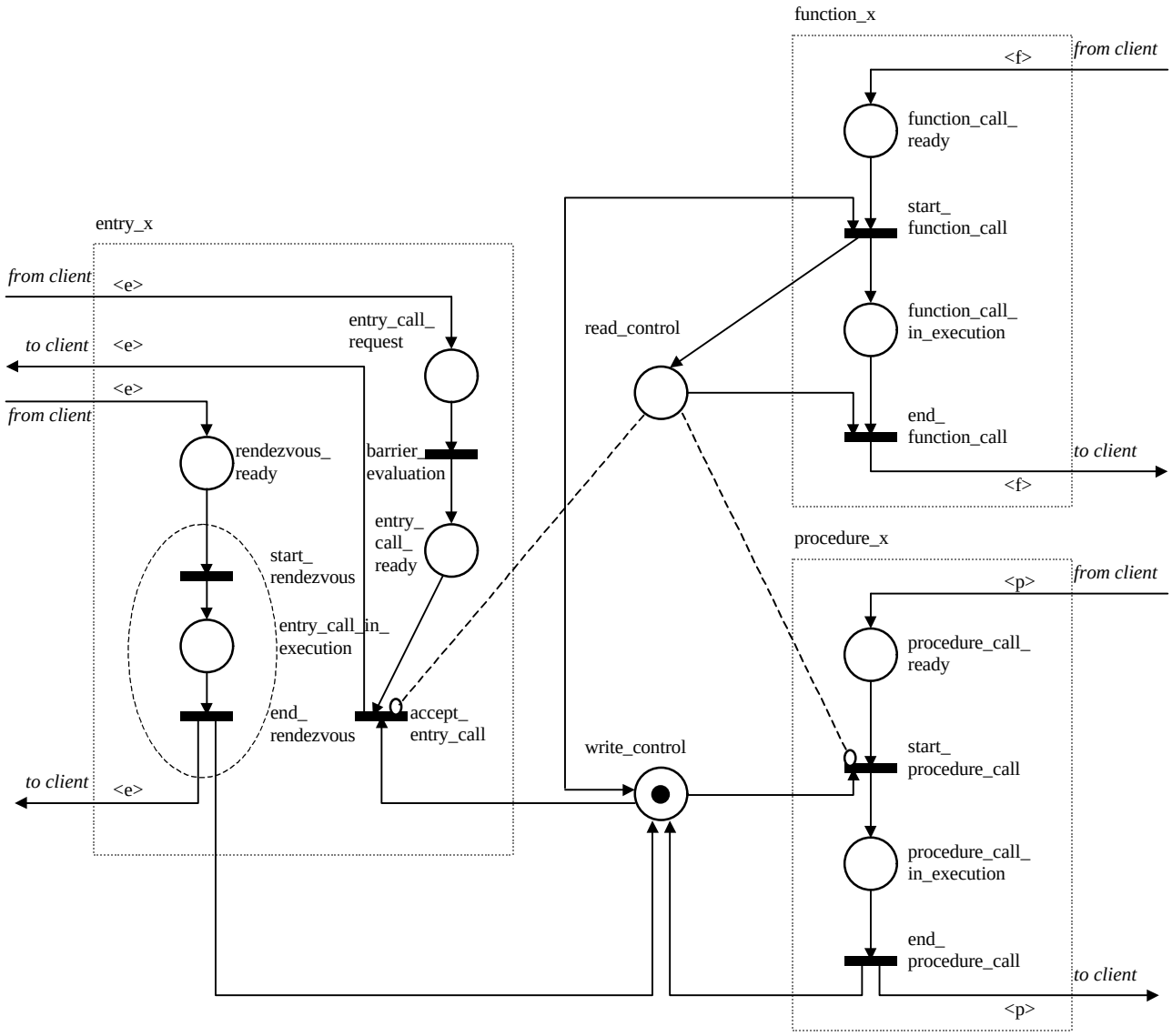


Figure 6. Protected object model

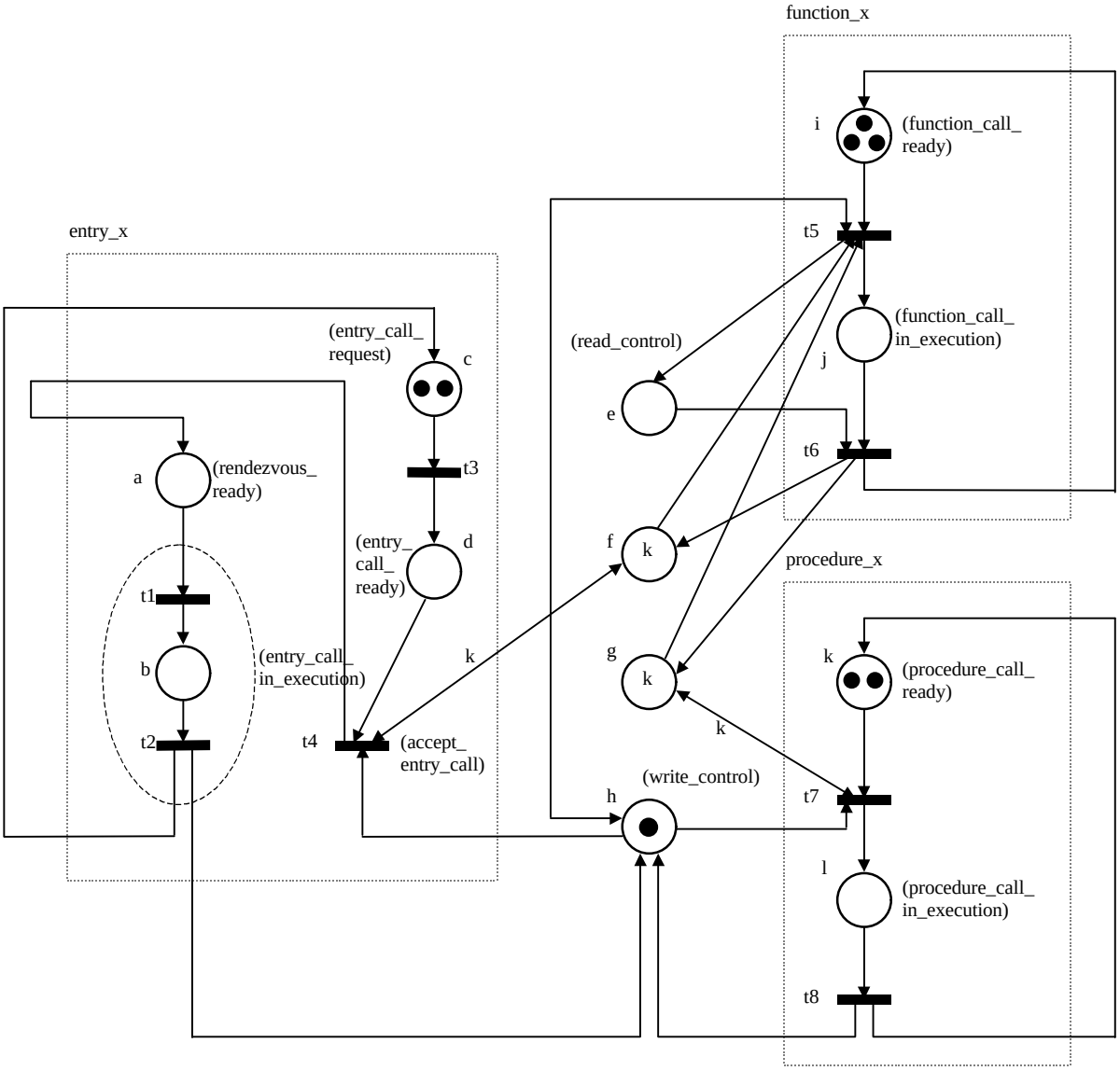


Figure 7. Transformed model of protected object

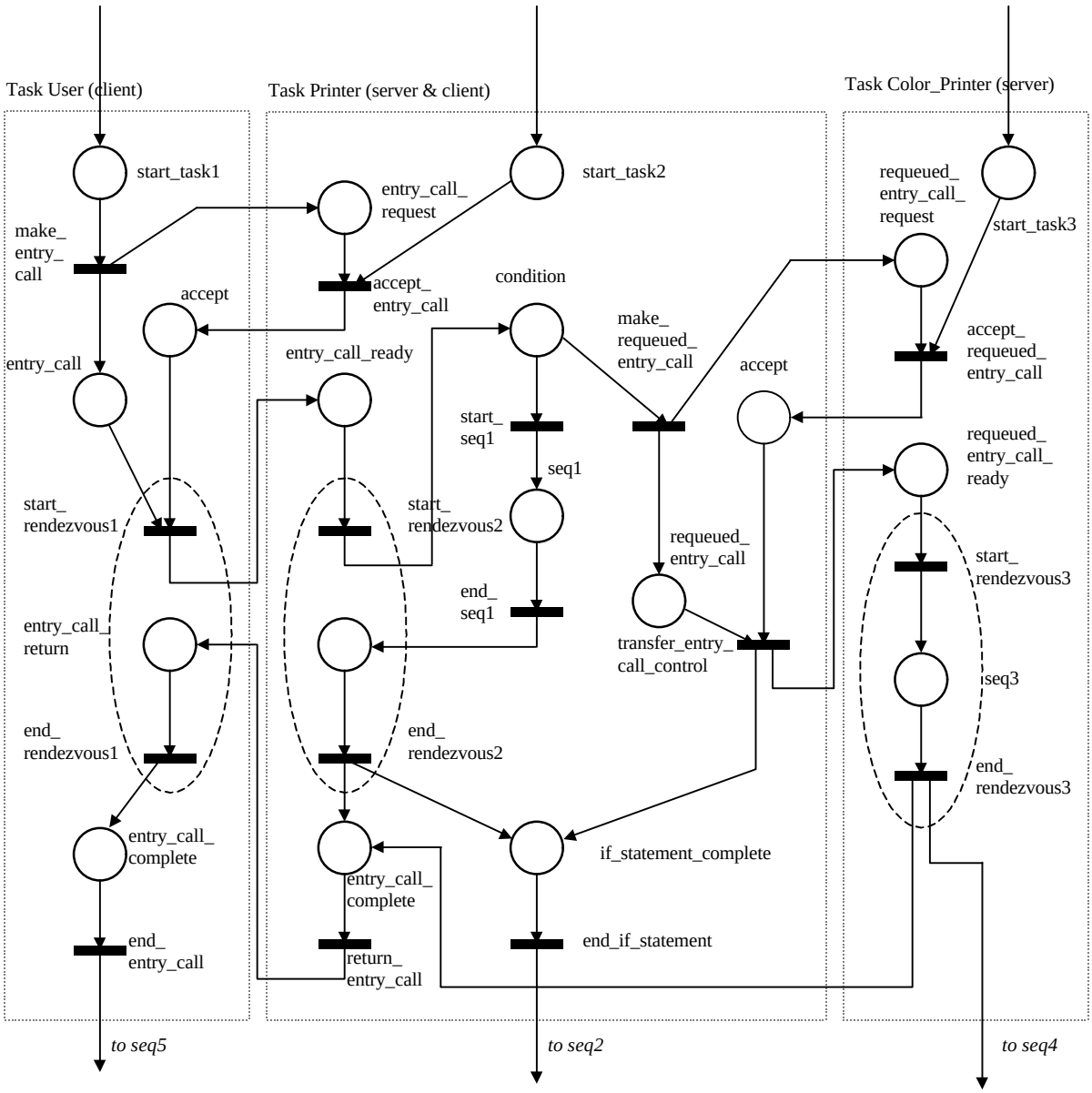


Figure 8. Requeue statement model

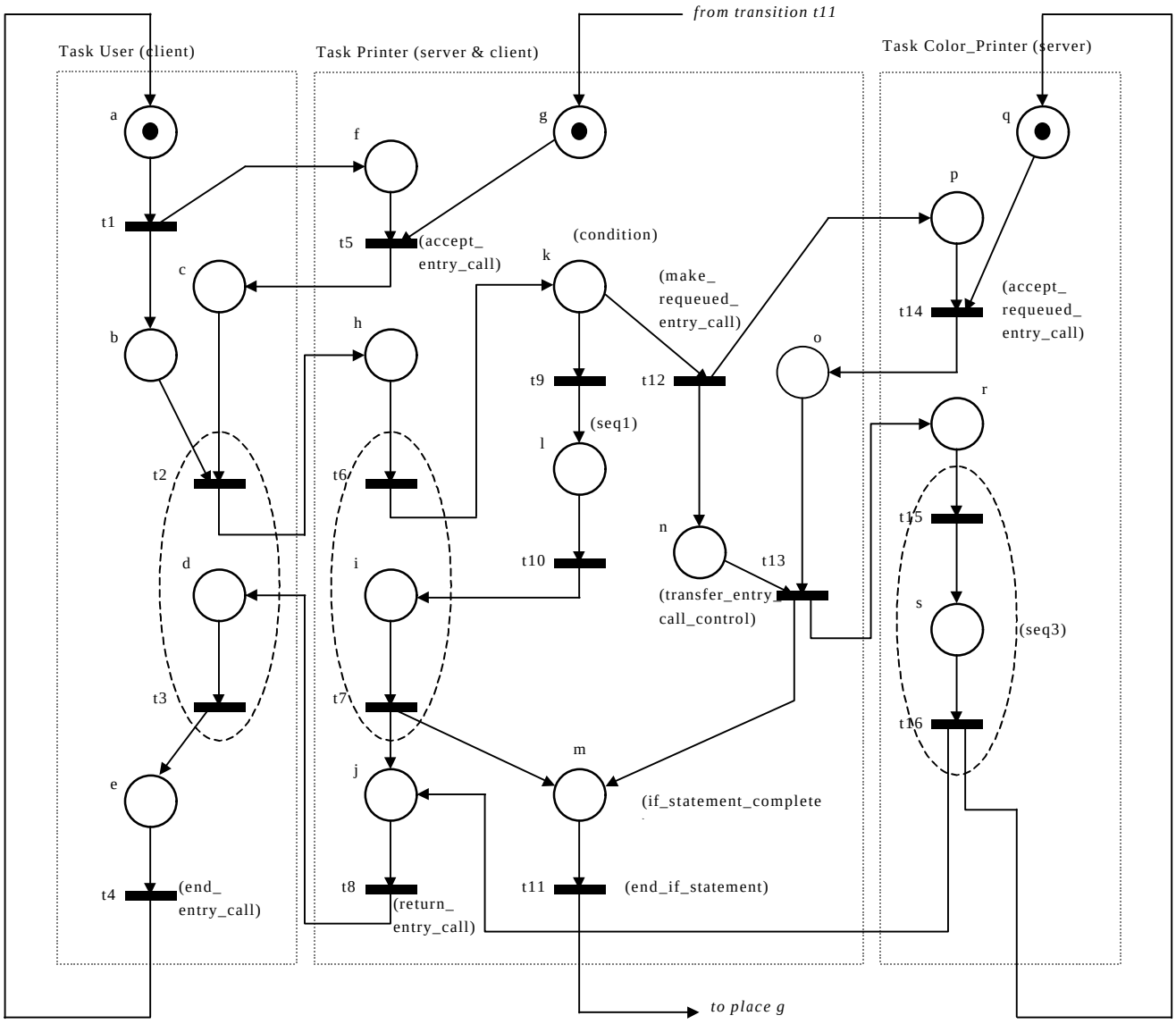


Figure 9. Tranformed model of requeue statement