A Specification and Verification Method for Preventing Denial of Service

CHE-FN YU, MEMBER, IEEE, AND VIRGIL D. GLIGOR, MEMBER, IEEE

Abstract—In this paper, we present a specification and verification method for preventing denial of service in absence of failures and of integrity violations. We introduce the notion of "user agreements" and argue that lack of specifications for these arguments and for simultaneity conditions makes it impossible to demonstrate denial-of-service prevention, in spite of demonstrably fair service access. We illustrate the use of this method with an example and explain why current methods for specification and verification of safety and liveness properties of concurrent programs do not handle this problem. The proposed specification and verification method is meant to augment current methods for secure system design.

Index Terms—Denial of service, liveness, safety, security, specification, temporal logic, verification.

I. INTRODUCTION

In previous work we have explained and illustrated the relationship between the denial-of-service problem and those of unauthorized disclosure and modification of information [3]. In particular, we have shown that denial of service can take place in absence of any type of system failure or integrity violation. In this paper, we argue that denial of service includes both safety and liveness properties of shared services in a distinct and heretofore unexplored manner.

To verify the absence of denial of service, a service specification model is introduced. A key component of that model is the separation of the service-sharing mechanism from the service-sharing policy. The need for specifying fairness and simultaneity conditions formally within the sharing policy is discussed. We argue that, in contrast with other properties, the prevention of denial of service requires specifications of service use; i.e., user agreements, which are external constraints on service invocations that must be obeyed by all service users. In general, these constraints cannot be converted into internal service-enforced constraints, such as those of the service-sharing mechanisms and policies. We show that the specification of sharing policy and that of user agreements form the basis for proof of denial-of-service prevention. We also explain why previous methods developed for verification of liveness and safety properties of concurrent programs cannot be used directly to demonstrate absence of denial of service in shared services.

Throughout this paper we assume that the definition of the denial of service is that presented in [1]-[3]. That is, a group of otherwise-authorized users of a specific service is said to deny service to another group of otherwise-authorized users if the former group makes the specified service unavailable to the latter group for a period of time that exceeds the intended (and advertised) waiting time. The notion of the service user is synonymous with that of a process that invokes the service through the service interface. The service waiting time can be specified explicitly through a maximum waiting time, as in the case of real-time service specifications, or implicitly by stating that any user will be granted access to the service eventually, i.e., in finite time. The latter waiting time specification is the one assumed throughout this paper. The conditions that specify the finite waiting time for a service define the finite waiting time (FWT) policy.

In this paper, we do not address denial-of-service problems that arise from failures of function or performance which might take place in the underlying mechanisms supporting a service implementation (e.g., communication delays, congestion, link failures, etc.). Nor do we address denial-of-service problems caused by integrity violations (e.g., unauthorized deletion or destruction of services and service objects). These problems are addressed by fault tolerance, performance, and access control models. Instead, we present a specification and verification method for preventing denial of service in absence of failures and of integrity violations. We argue that lack of specifications for user agreements and for simultaneity conditions makes it impossible to demonstrate denial-of-service prevention, in spite of demonstrably fair service access.

This paper consists of five sections and two Appendices. In Section II we explain the relationship between the denial-of-service problem and the safety and liveness properties of shared services. In Section III we present the specification and verification method and its underlying model. In Section IV we illustrate the method with an example of specification and verification of shared services. (Other examples, which include specification and verification of shared distributed services and proofs of
Denial-of-service prevention in Ada source code, are presented in [13].) Section V concludes this paper. Appendix A defines the temporal logic operators used in this paper, and Appendix B lists some useful temporal logic theorems.

II. DENIAL OF SERVICE VERSUS SAFETY AND LIVENESS PROBLEMS

Various safety and liveness properties of shared services or program modules have been widely used in the specification and verification of concurrent programs [9], [10], [12], and of communication protocols in computer networks [6]. These properties can also be used to verify that some denial-of-service problems have been eliminated from shared services. However, flawed specifications of concurrency control mechanisms and/or policies are not the only source of denial of service [1]. Also, safety and liveness problems that are unrelated to service sharing become irrelevant to denial of service.

A. Denial of Service as a Safety/Liveness Problem

Denial of service can be both a safety and a liveness problem. It takes place whenever one or both of the following situations occur: 1) some users prevent other users from making progress within the service for an arbitrarily long time; 2) some users cause other users to receive incorrect service by preventing the service from satisfying its intended functional specifications for the latter users, and thus the service is disabled in an unauthorized way. Denial-of-service instances of situation 1) are liveness problems, whereas denial-of-service instances of situation 2) are safety problems.

A user is said to make progress within a service if all of its service invocations will eventually terminate. Intuitively, a service allows its users to make progress whenever its sharing policies are fair, whenever it is free from deadlock, and whenever it is free from starvation. Fairness and freedom from starvation are liveness concerns. Furthermore, a service performs correctly whenever the different user operations within the service neither interfere with each other nor disable the service, and whenever the effects of a user operation within the service persist after the operation is committed by its user. All of these properties are safety concerns.

B. Denial of Service as a Distinct Safety/Liveness Problem

The major safety properties that have been formally specified and verified in the concurrent programming area: mutual exclusion, resource invariance, and deadlock freedom. The major liveness properties that have been formally specified and verified in a general way are termination and fairness. Starvation freedom is also a liveness property that has been verified formally in some examples. These safety and liveness properties are the fundamental properties of concurrency control mechanisms within shared services. Since services may be invoked concurrently, these properties must be used to prevent denial-of-service problems.

However, denial-of-service problems can also be caused by some other sharing problems unrelated to safety or liveness. Failure to use either adequate service-sharing mechanisms, such as resource quotas, or service-sharing policies, such as the finite waiting time for individual users, can also cause denial of service. For example, inadequate enforcement of resource quotas may stop the activity of the service [1], [2]; lack of finite-waiting-time specification may cause denial of service to some users even if a fair scheduling policy is enforced within the service [13].

One of the reasons why denial of service differs from the typical safety/liveness problems is that it may result from the inadequate use of the service. Undesirable user behavior can cause some users to receive incorrect service or prevent other users from making progress within the service. For example, the service behavior may depend on the order of entry operations invoked by users. Whenever a service assumes a certain invocation order and the assumed order is not guaranteed, then denial of service may take place. For example, a resource allocation service may expect a user to release resources some time after the resources have been acquired. However, a malicious user can monopolize the resources without releasing them for an arbitrarily long time. This user can thus deny the resources to other users. These types of denial of service may occur because shared services have no control over the users’ behavior outside the service. Although some aspects of the users’ behavior can be checked by other system facilities outside the service (e.g., through compilation checks), a resource can become unavailable for an arbitrarily long time when a user holding the resource gets blocked within another service; or when a user aborts before normal service termination and some of the service operations cannot finish their work on behalf of other users.

Notice that current service models, such as those based on monitors [8] and resource controllers [12], which are normally used for synchronization, are unable to protect against the denial-of-service problems caused by undesirable user behavior. This is because, although current service models can always schedule the proper order of operations among different users, they cannot always control the invocation order of individual users. Current service specification models focus on the integration of concurrency control into program modules of services and thus are irrelevant to the user behavior outside the service.

III. FORMAL SPECIFICATION AND VERIFICATION METHOD

In this section we introduce a specification and verification method that is suitable for the prevention of denial of service in shared services. We introduce the notion of "user agreement" and present a model for service and user-agreement specification. We argue that the specifi-
cation of fairness properties is insufficient to prevent de- nial of service, and that the specification of simultaneity conditions and user agreements are also necessary. To- gether these three specifications define the FWT policy.

A. The Notion of User Agreement

Denial of service can take place not only by exploiting flawed sharing mechanisms and/or policies within a ser- vice but also by undesired sequences of service invocation outside the service. Therefore, the allowed service-invocation sequences must be specified as constraints outside the service. Since these constraints include invocation sequences of multiple service users, they are called the "user agreements." User agreements for service access must be specified outside the service specification for two reasons. First, user agreements must eliminate the undesir- able interuser dependencies created by user misbehavior outside the service. Second, user agreements may refer to multiple services, and therefore cannot be included in any single service specification. For example, Haven- der's "ordered resource acquisition" approach to deadlock prevention [7] is an example of a user agreement that spans multiple resource-providing services inside an operating system.

The specification of user agreements outside a service does not necessarily mean that service users are trusted to obey them. Whenever compile-time checks on user code are impractical, the user agreements can be enforced by code outside the service that is executed before the service calls are actually issued. An example of such code in other areas, such as in the deadlock avoidance area, is that which determines whether a resource assignment to a user is safe [4].

To define user agreements for shared services, we have to analyze all possible invocation sequences that may be issued by each user and all possible invocation sequences that may invoke the shared service. For convenience, the former sequence is called the \( U \text{-seq} \) and the latter is called the \( S \text{-seq} \). Let \( U_i \text{-seq} \) be a partial order of service invocations issued by an individual user \( U_i \). Let \( S \text{-seq} \) be a partial order of concurrent service invocations by many users. Thus, \( S \text{-seq} \) is an invocation sequence that inter- leaves operations of individual \( U_i \text{-seq} \), and it preserves the original partial order of each \( U_i \text{-seq} \). Analysis of \( U \text{-seq} \) and \( S \text{-seq} \) allows us to define the user agreements for shared services.

1) Safe Service-Invocation Sequence: Operations needed for controlled service sharing can be classified into two categories: a) the resource-consuming operations and b) the resource-producing operations. For example, the \( Aquire \) and \( Release \) operations of resource allocators can be considered to be the consuming and producing operations, respectively. Similarly, the \( Put \) and \( Get \) operations of a bounded-buffer service are the consuming and pro-

\footnote{Other undesirable interuser dependencies, which can also be countered by user agreements, are those created by inadequate sharing mechanisms and policies [1].}
In this User seq, some resources allocated to user Ui are not relinquished before Ui terminates execution (i.e., there are more \( a_i \)'s than \( b_i \)'s). However, user Ui neither relinquishes resources before it acquires them, nor does it relinquish resources allocated to other users. Therefore, the user-invocation sequence is safe in the sense that it will not cause resource inconsistencies. For the same resource allocator, the sequences of User seqs denoted by \( a_i = a_i, b_i, a_i, b_i \) and \( a_i = a_i, b_i, b_i, a_i \) are not safe. The first User seq is not safe because user Ui attempts to relinquish one of the resources allocated to user Uj. The second User seq is not safe because user Ui attempts to relinquish its allocated resource twice. Both of these two User seqs may cause resource inconsistencies.

2) Live Service-Invocation Sequence: Let \( \alpha \) be a possible Service seq of a specific service. The elements of \( \alpha \) are service operations invoked by all users. The order of operations in \( \alpha \) is the real-time order of all service invocations. If \( a_i \) and \( b_i \) represent the consuming and producing operations respectively as defined above, then the sequence \( \alpha = a_1, a_2, a_3, b_1, a_4, a_5, b_2, b_3, \cdots \) is a possible Service seq. Here \( a_i^* \) represents a consuming operation of user Ui that is blocked within the service waiting for certain service conditions to become true. Service conditions are boolean functions of service states such as the resource state. The service conditions may change value only through invocations of the service-entry operations. For example, in the invocation sequence given above, the operation \( a_2 \), invoked by user \( U_3 \), is blocked for some resources to become available. After the resource consumed by user \( U_2 \) is relinquished, \( U_3 \) resumes its operation (since operation \( b_2 \) is immediately followed by operation \( b_3 \), and \( b_3 \) can appear in the Service seq only after \( U_3 \) resumes). For convenience, we use the following notation.

- \( p_i^*(c) \) is an operation \( p_i \) that is blocked for condition \( c \);
- \( p_i(c) \) is an operation \( p_i \) that resumes execution after being blocked for condition \( c \).

When condition \( c \) becomes true, \( p_i^*(c) \) might not become \( p_i(c) \) immediately due to the resumption of other blocked invocations that are waiting for the same condition \( c \). However, a condition may become true several times during the evolution of a specific Service seq. Given condition \( c \) for operation \( p_i \), let \( a_0 \) be the subsequence of a Service seq \( \alpha \) from the beginning of \( \alpha \) to the blocked operation \( p_i^*(c) \). Suppose that \( a_\ell \) represents the subsequence of \( \alpha \) between \( (j - 1) \)th and \( j \)th time that condition \( c \) becomes true after \( p_i^*(c) \). Then for operation \( p_i \) and the sequence \( \alpha = a_0, a_1, \cdots, a_\ell, b_\ell, \cdots \), we use the notation \( p_i^*(c) \rightarrow a_\ell \rightarrow p_i(c) \) to represent an invocation \( p_i \) that is blocked at the end of \( a_\ell \) and that resumes operation at the end of \( a_\ell \).

Definition: A service-invocation sequence Service seq is said to be live if for every blocked invocation \( p_i^*(c) \) there exists a set of subsequences \( a_0, a_1, \cdots, a_\ell, \alpha_j \) and \( \alpha = a_0, a_1, \cdots, a_\ell, \cdots \) such that \( p_i^*(c) \rightarrow a_\ell \rightarrow p_i(c) \).

For example, consider an allocator of a single resource type. Suppose that each type \( b \) operation makes condition \( c \) be true. If \( a_i = \text{Acquire} \), and \( b_i = \text{Release} \), then the sequence

\[
\alpha = a_1, a_2^*(c), b_1, a_3, \cdots, a_{2k-1},
\]

\[
a_2^*(c), b_{2k-1}, b_{2k}, \cdots
\]

is a live Service seq because \( a_2^*(c) \rightarrow a_i(c) \) for all \( k \geq 1 \), where \( a_2(c) \) occurs between \( b_{2k-1} \) and \( b_{2k} \). In contrast, the sequence

\[
\alpha = a_1, a_2^*(c), a_3^*(c), b_1, a_4^*(c),
\]

\[
b_3, \cdots, a_{i+1}^*(c), b_k, \cdots
\]

is not a live Service seq. This sequence has \( a_i^*(c) \rightarrow a_i(c) \) for all \( k \geq 3 \). However, for \( a_i^*(c) \), no \( a_j \) exists in \( \alpha \) such that \( a_j^*(c) \rightarrow a_i(c) \). Thus, operation \( a_i \) is blocked forever.

3) Analysis of User and Service Invocation Sequences: To obtain appropriate specifications for user agreements of a specific service, the analysis of both User seqs and Service seqs is required. Analysis that is limited to User seq is insufficient because User seqs provides only information about what users are allowed to do, but not what users must do. Furthermore, analysis that is limited to Service seq is also insufficient because liveness of a service-invocation sequence is meaningless without knowing that all completed operations in the Service seq have received the intended services.

Analysis of User seqs cannot determine the liveness property of the entire service-invocation sequence for at least two reasons. First, resources consumed by a user may not necessarily be produced later by the same user. For example, users that do a \( P \) operation on a certain semaphore may not do a \( V \) operation on the same semaphore, and vice versa. Thus, the availability of the resources protected by the semaphore cannot be determined by each individual User seq. Second, users may stop execution in the middle of their User seqs, and thus some operations, which other users may invoke, cannot finish their work on behalf of their users. For example, users' invocations may deadlock each other in several services. The occurrence of such deadlocks cannot be predicted by analyzing User seqs separately. Construction of user agreements based on live Service seqs solves the two problems described above.

Analysis of Service seqs helps establish liveness properties that cannot be provided by User seqs. However, a live Service seq does not guarantee that each individual operation receives its intended service. An invocation may return abnormally repeatedly (e.g., an exception is signaled before normal return), or may return normally with incorrect results whenever sharing control is incorrectly specified. Therefore, the analysis of both User seq and Service seq is required to determine appropriate user agreements of shared services. Of course, the appropriate User seq and Service seq depend on the sharing mechanisms and policies within the service under consideration.
B. The Specification Model for Shared Services

The service-specification model includes two major parts: the service specifications and the user-agreement specifications. The service specifications describe all the desired operations and properties that must be provided by the shared service. The user-agreement specifications describe all the desired properties that must be provided by the users of the shared service.

Given a specific service, the existence of undesirable interuser dependencies (and thus the potential for denial of service [1]–[3]) is determined by three major concerns:

- the service-sharing mechanisms and policies;
- the user-invocation sequence;
- the service-invocation sequence.

Appropriate internal service specifications are intended to eliminate undesirable interuser dependencies that may result from the first concern. Appropriate user-agreement specifications are used to eliminate undesirable interuser dependencies that may result from the second and the third concerns. The service-specification model separates sharing mechanisms from policy specifications because it distinguishes different types of service properties (i.e., safety versus liveness properties). We adopt a temporal-logic-based specification language [10], [11] to facilitate expressing the semantics of sharing mechanisms and sharing policies within a service and the user agreements for this service. (The semantics of temporal logic are reviewed in Appendix A.) We choose a specification method based on temporal logic to facilitate the construction of a service model for two reasons: 1) it has the power of reasoning about "future" events and thus is particularly suitable for expressing our notion of progress, and 2) it is convenient for expressing the semantics of the invocation order, and this order relation is an important part of our service model. However, other specification languages, not necessarily based on temporal logic, could be used with our specification method.

In the remainder of this section we explain how a specification that guarantees finite waiting for a service is written. We also explain the rationale for, and the specification of, the finite waiting time (FWT) policy. Then we introduce a specification for shared services that can be invoked concurrently and a specification of user agreements that describes properties for safe user-invocation sequences and live service-invocation sequences. Finally, the relationship between different specifications that constitute the FWT policy is presented. We also discuss their progress implications.

1) Specification of the FWT Policy: Access to shared services must be guaranteed to authorized users. Thus, an FWT policy must be adopted in the service to guarantee individual user progress within a shared service. For the purposes of this paper, the FWT policy consists of three different, yet mutually related specifications: fairness policies, simultaneity policies, and user agreements. The notion of the user agreement has been introduced earlier in this paper. The fairness and simultaneity policies are sharing policies specified within a service. The semantics of the fairness and simultaneity policies are presented in the next subsection, which also introduces the specification of shared services. Informally, the fairness policy states that a user will not get blocked forever within a service if that user has many opportunities to make progress. The simultaneity policy states that a user will eventually have all the opportunities needed to make progress within a service provided that the user agreements of that service can be satisfied.

Whenever a specification implies individual user progress within a service, then the FWT policy for that service is guaranteed. Thus, if the user agreements for that service can be satisfied, then the simultaneity policies guarantee the existence of progress opportunities for each user, and the fairness policies, in turn, guarantee that each user makes progress. The user agreements of a service can be satisfied in one or both of the following two ways depending on the service environment:

- apply service-invocation constraints to all service users so that all users obey the user agreements, and
- enforce user agreements so that the sequences of user invocations are regulated before the actual calls are issued.

Therefore, the fairness policy and the simultaneity policy plus the user agreements imply the FWT policy. Conversely, if any one of the specifications of fairness and simultaneity policies as well as the user agreements is not provided, then either the progress opportunities for some users may not always exist or the service may treat some users unfairly. Thus, the enforcement of the FWT policy cannot be guaranteed.

A FWT policy is best specified by using, whenever possible, internal service specifications; i.e., sharing-policy specifications. However, in general, it is impossible for internal service specifications to guarantee a finite waiting time. This assertion is based on the following line of reasoning. First, a finite waiting time for a service is guaranteed only if the service-invocation sequences of that service are live. However, a service specification cannot include the semantics of "live service-invocation sequences" because it cannot predict the users' behavior outside the service. Thus, we have to specify the properties required for live service-invocation sequences outside the service. Second, if we combined the fairness and the simultaneity policies into a single policy, then the resulting specification would not separate concerns properly and, in general, would become more complex. Such a specification would mix different properties and would become less comprehensible. This would make it difficult to implement the service from such a specification. For example, the fairness policy for users waiting within a service entry queue (e.g., the FIFO policy), and the simultaneity policy (e.g., any policy that prevents individual user starvation) are implemented by different liveness properties. Any specification that combines different, unrelated properties at the same level of abstraction would provide little specific information for practical implementation of any of the properties. For this reason, we decomp-
pose the FWT specification into different types of specifications.

2) Service Specification: A service specification defines the properties for concurrent service access and the necessary mechanisms and policies to enforce the desired properties. The skeleton of a service specification is shown in Fig. 1. Note the separation of the sharing mechanisms and policies. Below, we explain each keyword in Fig. 1.

• service: This keyword gives the name of the service.

• type: The data types that will be used in the service are specified. All specified types include a type name. If the type name is self-evident, no further explanation is given. If a given type is a construct of other types, it is specified explicitly. For example:

\[
\text{type } \text{userid} \\
\text{units} \\
\text{index} = 1 \ldots N \\
\]

• constant: The names of constant values or a group of structured constant values (e.g., constant array) are specified. An example of a specification for a constant is:

\[
\text{constant } \text{size : units} \\
\text{quota : array [userid] of units,} \\
\text{vid : userid, quota[id] \leq size} \\
\]

Note that specifications of constants may be exported through the interface specifications.

• variable: The names of variables are specified. Variables are used to express the states of a service, such as the state of the shared resource, the number of concurrent users, etc. A variable is specified by its name, type, and initial value, such as:

\[
\text{variable } \text{free : units, initially size} \\
\text{own : array [userid] of units,} \\
\text{initially vid : userid, own[id] = 0} \\
\]

• hidden operations: The hidden operations are those which are invisible to the users outside the service. Therefore, users cannot invoke hidden operations. Hidden operations can only be referenced from within the service. In general, they are used to help describe the behavior of internal operations. A hidden operation includes an operation name and the effects of the operation.

• interface operations: A user can access a service only by invoking the interface operations. An interface operation may include arguments. The identity of the invoking user is assumed to be an implicit argument inherent in every interface operation. The argument of an operation "op" is expressed as the construct: \text{op argument}. For example, in the resource allocator presented in Section IV below, \text{Acquire.id} represents the identity of the user process which invokes the "Acquire" operation currently. An interface operation includes an operation name (and arguments), exception conditions, and "effects" of the operation. Exception conditions describe the conditions under which the arguments will cause errors. The "effects" part describes the normal actions taken by the service when no exceptions occur. The variables within a service may change values after the execution of an interface operation. In order to distinguish the value of a service variable before modification from its value after modification, we add a symbol "'" to the right of the variable to represent its value after the operation has executed. For example, the "Acquire" operation of the resource allocator can be specified as follows:

\[
\text{interface operations} \\
\text{Acquire (n : units)} \\
\text{exception conditions : quota[id] \leq own[id] + n} \\
\text{effects : free' = free - n} \\
\text{own[id]' = own[id] + n} \\
\]

where \( n \) is the number of resources requested, \text{free} denotes the number of currently available resources, and \text{own[id]} specifies the number of resources currently assigned to the user \text{id}.

• resource constraints: The specification of "resource constraints" is essentially the same as that of "resource invariance" which has been widely used in the literature [5], [8]. Resource constraints give the properties of the service-provided resource and should always be true for all service states. For example, in the resource allocation service, one of the resource constraints is:

\[
\Box ((\text{free} \geq 0) \land (\text{free} \leq \text{size})) \\
\]

• concurrency constraints: Concurrency constraints specify the conditions under which concurrent operations are allowed to execute within a service. To specify such constraints, we use "#Active" to denote the number of concurrent active interface operations in the service and "#Active_op" to denote the number of concurrent active operations "op" in the service. An operation is said to be active if it is currently performing its computation for changing the resource state. For example, if a mutual exclusion is required between interface operations for
changing the resource state, and if an active operation will eventually terminate, then the concurrency constraints can be specified as follows:

**concurrency constraints**

$$\square(\#Active \leq 1)$$

$$(\#Active = 1) \iff (\#Active = 0)$$

- **fairness:** The fairness policy expresses the behavior required of a service such that the invocations of an operation which satisfy the necessary conditions infinitely often (i.e., repeatedly) will not be blocked forever. For example, in the resource allocation service, let condition $c_1$ denote the statement: "the number of resources currently available is no less than the number of resources requested," and let condition $c_2$ denote the statement: "there are no active operations in the service at this time." We need the following fairness policy: if both $c_1$ and $c_2$ can simultaneously become true infinitely often, then the blocked "Acquire" operation will eventually resume execution and finally terminate. This fairness policy can be specified as follows:

**fairness**

$$(\text{at(Acquire)} \land \square[\bigcirc(\text{free} \geq \text{Acquire.n}) \land (\#Active = 0)]) \iff \text{after(Acquire)}$$

- **simultaneity:** The simultaneity policy states that during the waiting period of an invocation, if every condition requested can be satisfied infinitely often, then all conditions eventually will be satisfied simultaneously. For example, let $c_1$ and $c_2$ be the conditions used in the description of fairness policy. The simultaneity policy of the resource allocation service can be defined by the following two specifications:

1) whenever an invocation to "Acquire" is blocked, if $c_1$ can be satisfied infinitely often and so can $c_2$, then $c_1$ and $c_2$ will eventually be satisfied simultaneously;
2) whenever an invocation to "Acquire" is blocked and some users always release their allocated resources repeatedly until $c_1$ becomes true, then $c_1$ will eventually become true.

In the second part, condition $c_1$ is further decomposed into a number of subconditions. Each of these subconditions denotes the statement: "one unit of the requested resource becomes available." We specify the simultaneity policy as follows:

**simultaneity**

$$(\text{in(Acquire)} \land \square[\bigcirc(\text{free} \geq \text{Acquire.n}) \land (\#Active = 0)]) \iff (\text{after(Acquire)}$$

$$(\text{in(Acquire)} \land \square[\bigcirc(\text{free} \geq \text{Acquire.n}) \land (\#Active = 0)]) \iff (\text{after(Acquire)}$$

3) **Agreement Specification:** As mentioned above, user agreements specify the properties that must be obeyed by the users of the shared services to prevent instances of denial of service. The skeleton of the agreement specification is shown in Fig. 2.

- **user agreement:** User agreements specify the allowed user operations, their order, and the required service-invocation sequence that must be obeyed by users outside the service. For example, to guarantee that all users of a resource allocators eventually make progress, the service-invocation sequences must preserve the liveliness property: whenever an "Acquire" operation gets blocked waiting for some resources to become available, then a sufficient number of "Release" operations must eventually become active and finally terminate until sufficient free resource units become available. Thus, the waiting "Acquire" operation would get a chance to make progress. This user agreement can be specified as:

**user agreement**

$$\text{in(Acquire)} \iff (\square[\bigcirc(\text{#Active.Release} > 0)]) \lor (\text{free} \geq \text{Acquire.n})$$

The user agreement is provided in an abstract form in the service-specification model and is valid for all services of the same class. However, the implementation of user agreements is strongly dependent on the service environment and thus may differ from service to service. To guarantee that a service-invocation sequence is live, users that share multiple classes of resource allocation services may be restricted to use only some allowed operation orders, viz. the "ordered resource" approach to deadlock prevention [7]. Alternatively, service users can be asked to claim the largest number of resource units of each service that the user will need at one time before any service access; viz. the "resource claim" approach to deadlock avoidance [4].
fairness only for invocations of the same interface operation. It would not guarantee fairness between different interface operations.

An important question is whether a fairness policy guarantees an individual user's progress. Fairness does not necessarily imply an individual user's progress because the opportunities for making progress may not always exist. Progress opportunities can only be provided by the application of simultaneity policies. Informally, a simultaneity policy states that a user will eventually have all the opportunities it needs to make progress if user agreements allow these opportunities to occur. However, a fairness policy together with a simultaneity policy still cannot guarantee an individual user's progress because the existence of such opportunities always depends on user agreements.

To illustrate the necessity for user agreements, let us consider the specification of a fairness policy. The conditions required for progress within a service are expressed as the hypothesis of the fairness policy. Some of these conditions can be satisfied only when the service-invocation sequence is live. Thus, in addition to specific sharing policies, some form of user-agreement specification is always required.

One may ask: if user agreements allow progress opportunities to be created, do we still need the simultaneity policy? A simultaneity policy is required because, although user agreements can allow progress opportunities to be created, they may not be able to guarantee that progress takes place within a service for at least two reasons. First, the occurrence of progress opportunities may also depend on other conditions that can be satisfied only by sharing mechanisms and policies of the service itself. For example, a service may require mutual exclusion among concurrent invocations. An operation invocation has an opportunity to make progress only when the currently active operation terminates. Termination would depend, in this case, upon the mutual exclusion conditions enforced within the service. Thus, the satisfaction of these conditions cannot be determined by the users outside the service. Second, even if the individual conditions requested are all dependent on user agreements, they may not be satisfied simultaneously. This is because other invocations, which are required to satisfy only part of these conditions, can make progress repeatedly. Thus, individual conditions requested by certain users may never become true simultaneously. The service invocations of these users will be blocked forever.

From the above discussion, it should be clear that the fairness policy, the simultaneity policy, and the user agreements are all required to guarantee individual progress.

IV. APPLICATION OF THE FORMAL SPECIFICATION AND VERIFICATION METHOD

In this section, we apply the model of shared service to specify a general resource allocator. The choice of this generic example is motivated by the fact that allocators provide a common source of denial-of-service problems in operating systems and communication protocols (failures and integrity violations notwithstanding). Further specification and verification examples, including distributed services such as network access, are presented in [13]. To demonstrate that the service model is appropriate for the prevention of denial of service, we will verify that all users of the given resource allocator eventually make progress and receive intended service.

A. Specification of a Resource Allocator

A resource allocator consists of a pool of resource units that can be shared by a group of users. Initially, the pool contains the total number of resource units. To prevent resource monopolization, the resource allocator maintains a resource quota for each user. The resource quota provides the maximum number of resource units that can be assigned to each user. Each user can acquire "n" units of the resource by invoking the "Acquire" operation. Similarly, each user can relinquish "n" units of the resource by invoking the "Release" operation. Based on the model of shared services, the service and agreement specifications of the resource allocator are shown in Figs. 3 and 4, respectively.

B. Formal Verification of the Resource Allocator

A resource allocator guarantees resource allocation to each user if all user invocations, which do not cause exceptions, eventually terminate and receive their intended service. We begin by proving that the specifications of the resource allocator guarantee that user invocations eventually make progress. We then show that the specifications ensure correct service for individual users. In the following proofs, we assume that user operations do not cause exceptions.

1) Progress Proofs: The term "invocations of resource allocator eventually terminate" can be expressed by the following two temporal formulas:

\[ at(Acquire) \rightarrow after(Acquire); \] (P1)

\[ at(Release) \rightarrow after(Release). \] (P2)

We must prove that given the service and user-agreement specifications of the resource allocator, both (P1) and (P2) are temporal theorems. To do this, we prove a series of lemmas based on the service and user-agreement specifications. To prove that (P2) is a temporal theorem, we first show that the resource allocator is repeatedly in a state in which no operations are in execution (Lemma 1). To prove that (P1) is a temporal theorem, first we show that a blocked "Acquire" invocation will eventually get a chance to proceed (Lemma 2). Then we show that if an "Acquire" invocation can be blocked forever, it should have an infinite number of chances to proceed (Lemma 3). Finally, we show that if (P1) is not true, then eventually the invocation of "Acquire" will be blocked for-
ever (Lemma 4). A list of derived temporal theorems, which are used in the following proofs, are given in Appendix B for reference.

**Lemma 1:** The formula $\Box (\#\text{Active} = 0)$ is a temporal theorem.

**Proof:** The concurrency constraints of the resource allocator (viz. Fig. 3) imply

$$\Box (\neg (\#\text{Active} = 0) \Rightarrow (\#\text{Active} = 0)).$$

From the derived theorem (D1) in Appendix B, we can conclude that

$$\Box (\#\text{Active} = 0).$$

**Theorem 1:** The formula $\text{at}(\text{Release}) \Rightarrow \text{after}(\text{Release})$ is a temporal theorem.

**Proof:** Applying Lemma 1 to the second fairness policy of the resource allocator (viz. Fig. 3), we complete the proof immediately.

**Lemma 2:** The formula $\text{in}(\text{Acquire}) \Rightarrow (\text{free} \geq \text{Acquire}.n)$ is a temporal theorem.

**Proof:** From the user-agreement specification of the resource allocator (viz. Fig. 4) and the temporal axiom (A5) in Appendix B, we have

$$\Box (\text{in}(\text{Acquire}) \Rightarrow (\text{free} \geq \text{Acquire}.n)) \Rightarrow (\Box (\#\text{Active}_\text{Release} > 0) \lor (\text{free} \geq \text{Acquire}.n)).$$

From the derived theorem (T3') in Appendix B, we have

$$\Box (\text{in}(\text{Acquire}) = (\Box (\#\text{Active}_\text{Release} > 0) \lor (\text{free} \geq \text{Acquire}.n))).$$

From the derived theorem (T2') in Appendix B, we have

$$\Box (\text{in}(\text{Acquire}) \Rightarrow (\Box (\#\text{Active}_\text{Release} > 0) \lor (\text{free} \geq \text{Acquire}.n))).$$

Hence,

$$\Box (\text{in}(\text{Acquire}) \Rightarrow (\Box (\#\text{Active}_\text{Release} > 0) \lor (\text{free} \geq \text{Acquire}.n)) \Rightarrow (\Box (\#\text{Active}_\text{Release} > 0) \lor (\text{free} \geq \text{Acquire}.n)).$$

From the temporal formulas (L1), (L2), and the derived theorem (D3) in Appendix B, we obtain

$$\Box (\text{in}(\text{Acquire}) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n))).$$

**Lemma 3:** The formula $\Box \Box \text{in}(\text{Acquire}) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n))$ is a temporal theorem.

**Proof:** From Lemma 1, the first simultaneity policy of the resource allocator (viz. Figure 3) implies

$$(\text{in}(\text{Acquire}) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n))) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n)).$$

From the derived theorem (D4) in Appendix B, we have

$$(\Box (\text{in}(\text{Acquire}) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n))) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n))).$$

From the derived theorems (T4), (T1), and (T2') in Appendix B, we have

$$(\Box (\text{in}(\text{Acquire}) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n))) \Rightarrow (\Box (\text{free} \geq \text{Acquire}.n)).$$

Also, Lemma 2 implies

$$\Box (\text{in}(\text{Acquire}) \Rightarrow \Box (\text{free} \geq \text{Acquire}.n)).$$
From the temporal formulas (L3), (L4), and the derived theorem (D2) in Appendix B, we can conclude that
\[ \Diamond \Box in\text{(Acquire)} \Rightarrow \Box \Diamond ((\text{free} \geq \text{Acquire.n}) \land (\#Active = 0)). \]

**Lemma 4:** If the temporal formula \( \Diamond (at(\text{Acquire}) \land \Box (\neg (after(\text{Acquire})))) \) is true, then the temporal formula \( \Diamond \Box (in(\text{Acquire})) \) is also true.

**Proof:** From the semantics of the control predicates \( at, after, \) and \( in \) given in Appendix A, we have
\[ \Diamond (at(\text{Acquire}) \land \Box (\neg (after(\text{Acquire})))) \Rightarrow \]
\[ \Diamond (at(\text{Acquire}) \land \Diamond (in(\text{Acquire}))) \Rightarrow \]
\[ \Diamond \Box (in(\text{Acquire})). \]

**Theorem 2:** The formula \( at(\text{Acquire}) \Rightarrow after(\text{Acquire}) \) is a temporal theorem.

**Proof:** Suppose that the formula is not a temporal theorem. Then
\[ \Diamond (at(\text{Acquire}) \land \Box (\neg (after(\text{Acquire})))) \].

From Lemmas 3 and 4 we obtain
\[ \Box \Diamond ((\text{free} \geq \text{Acquire.n}) \land (\#Active = 0)). \]

From the first fairness policy of the resource allocator (viz. Fig. 3) and this temporal formula we obtain \( at(\text{Acquire}) \Rightarrow after(\text{Acquire}) \) which is supposed to be false. This completes the proof.

2) **Service Correctness Proofs:** To demonstrate that user invocations receive correct service when they terminate, we show that the following temporal formulas are temporal theorems.

\[ \forall id\ ((own[id] = M) \land \Diamond (after(\text{Acquire}(n)))) \Rightarrow \]
\[ \Diamond (own[id] = M + n), \quad (C1) \]

\[ \forall id\ ((own[id] = M) \land \Diamond (after(\text{Release}(n)))) \Rightarrow \]
\[ \Diamond (own[id] = M - n), \quad (C2) \]

\[ \forall id\ ((own[id] = M) \land \Box (\neg (after(\text{Acquire}))) \land \neg (after(\text{Release}))) \Rightarrow \]
\[ \Box (own[id] = M). \quad (C3) \]

The proofs are rather straightforward and can be obtained by using the specification of sharing mechanisms.

**Theorem 3:** The temporal formulas (C1), (C2), and (C3) are theorems.

**Proof:** First, from the concurrency constraints of the resource allocator (viz. Fig. 3), concurrent invocations are required to be mutually exclusive. Thus, if \( \Diamond (after(\text{Acquire}(n))) \), then the “effects” part of “Acquire” operation implies (C1) directly. Similarly, the “effects” part of “Release” operation implies (C2) directly.

Second, applying the temporal axiom \( (P \text{ UNTIL } Q) = (\Box \neg Q \Rightarrow \Box P) \), (C3) is directly implied by the fourth resource constraint (viz. Fig. 3).

**V. Discussion**

We have presented here a service model for the prevention of denial of service in computer systems. Based on this model we specified shared services using temporal logic. The advantage of using this model and specifications method to solve safety and liveness problems which cause denial of service is that essential service properties are stated based on the concept of abstraction. The service model allows us to interpret, specify, and verify service properties for preventing denial of service in different service environments. This is more important than it might appear because denial of service usually appears in multiple guises in practice [1].

Other research work also used abstract specification methods to describe required properties [5], [9], [12]. However, these methods are not particularly suitable for the prevention of denial-of-service problems for the following reasons. First, they do not provide specifications of simultaneity policy within the service. (Although in principle these methods could provide simultaneity-policy specifications, they currently do not include such specifications.) Second, they do not specify properties that must be satisfied by the users outside the service. (Instead, such properties usually appear as hypotheses of fairness policies specified within a service. However, user behavior outside the service may make these hypotheses false. Therefore, external service specifications are still necessary.)

To date, only the fairness policy has been specified and verified in a general way. However, the fairness policy cannot prevent denial-of-service instances caused by conspiracy among a group of users that manage to monopolize shared resources. Our service specification and verification method provides a formal semantics of simultaneity policies that are applicable to all shared services. Simultaneity policies are necessary but are not sufficient to guarantee individual user progress within a service.

The main reason that the current specification methods are unsatisfactory is that they only attempt to express the properties that must be enforced within a service. User-agreement specifications, which are necessary to eliminate the effects of undesirable interuser dependencies within the service, are usually not provided. In this case, the denial-of-service problems cannot be solved by the direct application of current specification methods.

**APPENDIX A**

**SEMANTICS OF TEMPORAL LOGIC**

The use of temporal logic in program specification and verification was first proposed by Pnueli [11]. In general, a temporal logic formula is constructed from a set of predicates, the usual logic operators, and the temporal operators \( \Box, \Diamond, \) and UNTIL. A predicate is a boolean function of a computation state. The unary operator \( \Box \) is pronounced “henceforth.” If \( P \) is a predicate, the formula \( \Box P \) means “\( P \) is true now and will remain true for all future states in the computation.” The unary operator \( \Diamond \) is pronounced “eventually.” The formula \( \Diamond P \) means “\( P \) is true now or will become true sometime in the
future." With these unary temporal operators, many important properties of concurrent programs can be stated. For example, the progress property of an operation \( \text{op} \) can be expressed by the formula:

\[
at \text{(op)} \Rightarrow \diamond \text{after} \text{(op)},
\]

where \( at \text{(op)} \) and \( \text{after} \text{(op)} \) are predicates used to keep track of the control position of an invocation. The predicate \( at \text{(op)} \) is true if and only if control is at the entry point of \( \text{op} \), and \( \text{after} \text{(op)} \) is true if and only if control is at the exit point of \( \text{op} \). The predicate \( \text{in} \text{(op)} \) must also be introduced. The predicate \( \text{in} \text{(op)} \) is true if and only if control is anywhere inside the operation \( \text{op} \), including its entry point, but excluding the exit point. Hence, if control is currently at the entry point to \( \text{op} \) and never reaches the exit point thereafter, then control will remain in \( \text{op} \) forever; i.e.,

\[
(at \text{(op)}) \land \Box \neg \text{after} \text{(op)}) \Rightarrow \Box \text{in} \text{(op)}.
\]

Another useful temporal formula, which states that a property \( P \) always causes another property \( Q \) to become true subsequently, can be expressed by

\[
P \rightarrow Q \equiv \Box (P \Rightarrow Q).
\]

The combination of these two unary temporal operators is also useful. For example, to express that a property \( P \) is satisfied "infinitely often," we can use the formula \( \Box P \) (infinitely often \( P \)). The \( \Box \) operator is especially useful for expressing the fairness property within a service whenever concurrent access is allowed.

The binary temporal operator UNTIL is used to express relationships between two points in a computation (i.e., to express an ordering property). The formula \( P \text{ UNTIL } Q \) means "\( P \) is true for all states until the first state where \( Q \) is true;" i.e.,

\[
P \text{ UNTIL } Q \equiv P \text{ remains true until } Q \text{ becomes true}.
\]

### Appendix B
**Derived Temporal Theorems**

This Appendix presents the syntax of temporal formulas and derives a list of temporal theorems that are used in this paper. Some portions of this Appendix have been taken from Hailpern's report [5].

#### A. The Syntax of Temporal Formulas

A description of a temporal logic system includes four parts: a list of atomic predicates; a set of formation rules that define which predicates (expressed in terms of the atomic predicates) are (well-formed) formulas; a set of distinguished formulas, known as axioms; and a set of inference rules that permit operations on axioms and on those formulas that have been derived from previous applications of the inference rules. The formulas obtained by applications of inference rules are known as theorems. If formula \( P \) is an axiom or a theorem, then we write \( \vdash P \). The syntax of temporal formulas is based on the following rules and axioms.

1) **Formation Rules:**
- an atomic predicate is a formula;
- if \( A \) is a formula, then so are \( \neg A, \Box A, \) and \( \Diamond A \);
- if \( A \) and \( B \) are formulas, then so are \( (A \lor B), (A \land B), (A \Rightarrow B), \) and \( (A \Leftarrow B) \).

2) **Temporal Axioms:** Let \( P \) and \( Q \) be formulas. The following temporal formulas are axioms:

\[
\begin{align*}
(A1): \& P \Rightarrow \neg \Box \neg P, & (A1'): \& P \Rightarrow \Diamond \neg P; \\
(A2): \Diamond P = P, & (A2'): \Box P = \Diamond P; \\
(A3): \Box (P \Rightarrow Q) = (\square P \Rightarrow \Box Q), & (A3'): (\Box P \land \Diamond Q) = \Diamond (P \land Q); \\
(A4): (P \text{ UNTIL } Q) = (\Box Q \Rightarrow \Box P), & (A5): (P \Rightarrow Q) \equiv \Box (P \Rightarrow \Diamond Q).
\end{align*}
\]

3) **Interface Rules:**

\[
\begin{align*}
(I1): \vdash \neg P & \Rightarrow \neg \Box \neg P; \\
(I2): \vdash \Box (P \equiv Q) & \Rightarrow \Box (f(P) \equiv f(Q)).
\end{align*}
\]

#### B. Derived Theorems

1) **Basic Derived Theorems:** The following theorems have been proven in [5]:

\[
\begin{align*}
(T1): \Box \Box P = \Box P, & (T1'): \Diamond \Diamond P = \Diamond P; \\
(T2): \Box \Box \Box P = \Box \Box P, & (T2'): \Box \Diamond \Box P = \Box \Diamond P; \\
(T3): \Box (P \land Q) = (\Box P \land \Box Q), & (T3'): \Diamond (P \text{ UNTIL } Q) = \Diamond (P \land Q}; \\
(T4): \Box (P \lor Q) = (\Box P \lor \Box Q), & (T4'): \Diamond (P \lor Q) = (\Diamond P \lor \Diamond Q); \\
(T5): (\Box P \lor \Box Q) = (\Box P \lor \Box Q), & (T5'): (\Diamond P \lor \Diamond Q) = (\Diamond P \lor \Diamond Q); \\
(T6): (\Box P \lor \Box Q) = (\Box P \lor \Box Q), & (T6'): (\Diamond P \lor \Diamond Q) = (\Diamond P \lor \Diamond Q); \\
(T7): (\Box P \lor \Box Q) = (\Box P \lor \Box Q), & (T7'): (\Diamond P \lor \Diamond Q) = (\Diamond P \lor \Diamond Q); \\
(T8): (P \Rightarrow Q) = ((P \lor R) = (Q \lor R)); & (T9): (P \text{ UNTIL } Q) = (\Box P \Rightarrow \Diamond Q). \\
\end{align*}
\]

2) **Other Derived Theorems:** The theorems listed below are used in Section IV of this paper. Each theorem is followed by its proof.

\[
\begin{align*}
(D1): \Box (\neg P \Rightarrow \Diamond P) & \Rightarrow \Box \Diamond P; \\
(D2): (\langle P \Rightarrow Q \rangle \land (\langle P \land Q \Rightarrow R \rangle)) & = (P \Rightarrow R) \land (P \Rightarrow Q); \\
(D3): (\neg P \land (P \lor Q)) & \land (\neg P \lor (P \Rightarrow Q)) \equiv (\neg P \land (P \lor Q)) \land (\neg P \lor (P \Rightarrow Q)); \\
(D4): (P \Rightarrow Q) \land (\neg P \lor (P \Rightarrow Q)) & \equiv (\neg P \lor (P \Rightarrow Q)) \land (P \Rightarrow Q).
\end{align*}
\]
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) =\)
\((\neg P \lor (Q \land R))\)  
\((\neg P \lor (Q \land R)) \land ((P \land Q) \Rightarrow R) =\)
\((\neg P \lor (Q \land R))\)  
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) =\)
\((\neg P \lor (Q \land \neg P \lor R))\)  
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) =\)
\((\neg P \lor (Q \land \neg P \lor R))\)  
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) =\)
\((\neg P \lor (Q \land \neg P \lor R))\)  

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)

\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)
\((P \Rightarrow Q) \land ((P \land Q) \Rightarrow R) \Rightarrow \Box (P \Rightarrow R)\)