Task Structures as a Basis for Modeling Knowledge-Based Systems

Jonathan Lee*
Department of Computer Science and Information Engineering,
National Central University, Chungli, Taiwan

Recently, there has been an increasing interest in improving the reliability and quality of AI systems. As a result, a number of approaches to knowledge-based systems modeling have been proposed. However, these approaches are limited in formally verifying the intended functionality and behavior of a knowledge-based system. In this article, we proposed a formal treatment to task structures to formally specify and verify knowledge-based systems modeled using these structures. The specification of a knowledge-based system modeled using task structures has two components: a model specification that describes static properties of the system, and a process specification that characterizes dynamic properties of the system. The static properties of a system are described by two models: a model about domain objects (domain model), and a model about the problem-solving states (state model). The dynamic properties of the system are characterized by (1) using the notion of state transition to explicitly describe what the functionality of a task is, and (2) specifying the sequence of tasks and interactions between tasks (i.e., behavior of a system) using task state expressions (TSE). The task structure extended with the proposed formalism not only provides a basis for detailed functional decomposition with procedure abstraction embedded in, but also facilitates the verification of the intended functionality and behavior of a knowledge-based system. © 1997 John Wiley & Sons, Inc.

I. INTRODUCTION

Recently, there has been an increasing interest in improving the reliability and quality of AI systems.1 As a result, a number of approaches to knowledge-based systems modeling have been proposed in the Software Engineering and Artificial Intelligence literature, e.g., Refs. 2 to 8.

In particular, Tsai, Slagle, and their colleagues have conducted an empirical case study on requirements specifications of knowledge-based systems.4,6 These approaches are important in demonstrating the feasibility of developing specifications for knowledge-based systems. Meanwhile, Chandrasekaran and several other researchers have advocated using task structures as a general modeling

*YJLEE@SE01.CSIE.NCU.EDU.TW
approach to knowledge-based systems. The task structure is a tree of tasks, methods, and subtasks, which provides a framework for organizing knowledge in knowledge-based systems in a step-wise refinement fashion. However, these approaches are limited in formally verifying the intended functionality and behavior of a knowledge-based system.

Our emphasis in this article is to provide a formal treatment to task structures to formally specify and verify knowledge-based systems modeled using these structures. The specification of a knowledge-based system modeled using task structures has two components: a model specification that describes static properties of the system, and a process specification that characterizes dynamic properties of the system. The static properties of a system are described by two models: a model about domain objects (domain model), and a model about the problem-solving states (state model). The dynamic properties of the system are characterized by (1) using the notion of state transition to explicitly describe what the functionality of a task is, and (2) specifying the sequence of tasks and interactions between tasks (i.e., behavior of a system) using task state expressions (TSE).

TSE extends regular expressions with a task-based composition operator to combine expressions at different levels in a task structure, and a follow operator to describe a partial system behavior. TSE can be associated with a task to describe a global interaction or with a method to specify local task sequencing in the structure. Progression operators and frame axioms are adopted for constructing the descriptions of a state through a TSE to an initial state description in the context of task structures. The task structure extended with the proposed formalism not only provides a basis for detailed functional decomposition with procedure abstraction embedded in, but also facilitates the verification of the intended functionality and behavior of a knowledge-based system.

In the sequel, we first describe some background work in task structures in the next section. The formal foundation of task structures is proposed in Section III. We discuss how to propagate state descriptions in the context of task structures in Section IV. Verification of specifications modeled using the structures is presented in Section V. In Section VI, examples based on R1/SOAR are given to demonstrate the benefits of our approach. A discussion on the comparison of the proposed approach to related work is in Section VII. Finally, we summarize the proposed approach and outlined the plan for our future research.

II. BACKGROUND WORK IN TASK STRUCTURES

Variations of task structures have been proposed, e.g., Refs. 2, 5, and 12 (Fig. 1). Steels [see Fig. 1(a)] has proposed work along lines that build on the notion of tasks and task structures. In his formulation, the task structure is intended to specify the task/subtask decomposition of a complex task. There is no explicit notion of alternate methods, that is, the method [M in Fig. 1(a)] is chosen implicitly in the analysis. The task sequencing relationship in a method cannot be specified.

In Ref. 2, Chandrasekaran has described a task structure in which we can explicitly identify alternative methods for each tasks in a task structure. A method
can set up subtasks, which themselves can be accomplished by various methods. Only search control knowledge can be described, for instance, the task sequencing relationship between subtasks B and C in Figure 1(b).

Hofsted et al.\textsuperscript{12} has presented a task structure diagram, in which tasks can be recursively defined in terms of subtasks, and can be decomposed into a hierarchy of subtasks. Decisions are used if performing a task involves choices between subtasks [denoted as a circle in Fig. 1(c)]. In task structure diagrams, sequential execution, iteration, choice, parallelism, and synchronization can be expressed. For example, task sequencing relationship among subtasks B, C, D, and E is graphically documented in the task structure diagram, namely, task B is executed first, which is followed by the task C and decision $d_i$ in parallel fashion, and in turn is followed by the choice between task D or E. Similar to Steel’s approach, there is no explicit notion of alternative methods. The local task sequencing is expressed using the task structure diagram.

Problems with these approaches can be summarized below:

- cannot verify the consistency of system behaviors due to the lack of a formal foundation of task structures (and tasks);
- lack of a mechanism that can describe partial system behavior, and can fill in the missing detail as the information is available;
- unable to describe global interactions between tasks at different levels.

In the next section, we propose a formal treatment of task structures to formally specify functional specifications and dynamic behaviors of knowledge-based systems modeled using task structures.

**Figure 1.** Variations of task structures.
III. A FORMAL FOUNDATION OF TASK STRUCTURES

The specification of a knowledge-based system modeled using task structures has two components: a model specification that describes static properties of the system, and a process specification that characterizes dynamic properties of the system. The static properties of a system are described by two models: a model about domain objects (domain model), and a model about the problem-solving states (state model). The dynamic properties of the system are characterized by (1) using the notion of state transition to explicitly describe what the functionality of a task is, and (2) specifying the sequence of tasks and interactions between tasks (i.e., behaviors of a system) using task state expressions (TSE). In this article we will focus on aspects in the proposed framework that facilitate the process specification and verification of a knowledge-based system's functional requirements. More detailed descriptions about model specifications can be found in Refs. 8 and 13. In this section, we first formulate the notion of tasks in task structures, provide a formalism, task state expressions, to describe the relationships among tasks, methods, and subtasks, and conclude with the specifications of tasks and methods.

A. Formal Semantics of Tasks in Task Structures

To explicitly specify what a task is in a task structure, a task is treated as (1) a state-transition which can take one state to another through a sequence of intermediate states, and (2) a functional unit that can be specified by using preconditions, protections, and postconditions, which are partial state descriptions about the state before, during, and after performing the task. State transition model is thus adopted for the purpose of formulating and analyzing the proposed specification methodology. In the general form of the model (e.g., Ref. 14), the world is viewed as being in one of a potentially infinite number of states. The effect of a task is to cause the world to make a transition from one state to another. Each task is therefore modeled by a set of triple \((b, d, a)\) where \(b\) and \(a\) are states, and \(d\) is a sequence of states, \(b\) being the “before state,” \(d\) being the “during state,” and \(a\) being the “after state.”

**Definition 1** (State Transition): A task \(T\) is defined as a set of triples \((b, d, a)\), where \(b\) and \(a\) denote the state before and after an invocation of \(T\), and \(d\) denotes the sequence of states during the invocation of \(T\).

Formulas in first-order logic are used for state descriptions. The precondition of a task is a condition (i.e., a formula) that needs to be true for all states before the task. The rigid postcondition of a task is a condition that is true for all states after the task. The protection of a task is a condition that needs to be held before, during, and after the execution of the task. These are formally defined below.

**Definition 2** (Functional Specification): Let a task \(T\) be a set of triples \((b, d,a)\).
1. A formula \( \varphi \) is a precondition of a task \( T \), denoted as \( \mathcal{P}_T \), if and only if \( \varphi \) is true in state \( b \), for every triple \( (b, d, a) \in T \).

2. A formula \( \varphi \) is a protection of a task \( T \), denoted as \( \mathcal{R}_T \), if and only if for every triple \( (b, d, a) \in T \), \( \varphi \) is true in state \( b \), \( \varphi \) is true in state sequence \( d \), and \( \varphi \) is true in state \( a \).

3. A formula \( \varphi \) is a rigid postcondition of a task \( T \), denoted as \( \mathcal{R}_T \), if and only if for every \( (b, d, a) \in T \), \( \varphi \) is true in state \( a \).

4. A formula \( \varphi \) is a soft postcondition of a task \( T \), denoted as \( \mathcal{B}_T \), if and only if for every \( (b, d, a) \in T \), the degree to which \( \varphi \) is true in state \( a \) is a real number \( \in [0, 1] \).

### B. Task State Expression

Task state expression (TSE), based on regular expressions, is to describe the dynamic behavior of a system modeled using task structures. Regular expression has been used in AI planning to represent the procedural knowledge (e.g., Ref. 15) and in software engineering to describe the behavior of a software system (e.g., Ref. 16). A TSE is an expression that defines (1) the desirable sequence of tasks that are expected to have processed before a given task, and (2) interactions between tasks at different levels. The notion of conditional expression is also adopted in TSE. We formally define TSE below.

**Definition 3 (Task State Expression).** The task state expression, \( \varepsilon \), over a set of tasks, \( \mathcal{F} = \{T_1, T_2, \ldots , T_n\} \), is defined as follows:

1. \( \varepsilon \) is a task state expression, where \( \varepsilon \) stands for null.
2. For each task \( T \in \mathcal{F} \), \( (T) \) is a task state expression.
3. If \( \varepsilon_i \) and \( \varepsilon_j \) are task state expressions, then \( (\varepsilon_i, \varepsilon_j) \) is a task state expression, where ``\( , \)'' denotes immediately follow.
4. For each expression \( \varepsilon_i, \beta \varepsilon \) is a task state expression, where \( \beta \) is a well-formed formula to be attached to a TSE, \( \varepsilon \), to represent the conditional branching. For a TSE, \( \beta \varepsilon \) means ``If \( \beta \) then \( \varepsilon \) else if \( \beta \) then \( \varepsilon \) ... else if \( \beta \), then \( \varepsilon \)'' where \( \lor \) denotes selection, and \( \beta \), \( \beta \), ... \( \beta \) are mutually exclusive.
5. For each expression \( \varepsilon_i, [\varepsilon] \) is a task state expression and is a short hand notation for \( (\varepsilon \lor \varepsilon) \), where ``\( [ \] \)'' denotes optional.
6. If \( \varepsilon_i \) and \( \varepsilon_j \) are task state expressions, then \( (\beta \varepsilon_i)* \) and \( (\varepsilon_i, \beta \varepsilon_j)* \) are task state expressions, where ``\( * \)'' denotes iteration and ``\( \beta \)'' is an iteration condition.
7. If \( \varepsilon_i \) and \( \varepsilon_j \) are task state expressions, then \( (\beta, \varepsilon_j) \) are task state expressions, where ``\( \beta \)'' denotes follow.

Graphical notations of TSE operators in a task structure diagram are shown in Figure 2. The relationship between subtasks \( B \) and \( C \) of method \( M_t \) is a sequential relationship [that is, \( (B, C) \)]. Selectional relationship is illustrated using the relationship between the subtasks of method \( M_5 \), denoted by conditions \( \beta_i, \beta_j \), and a ``\( \lor \)'' within a rectangle [e.g., \( (\beta_i D \lor \beta_j E) \)]. As for iteration, there are two cases: (1) the iteration condition is associated with the whole expression (namely, \( \beta_i(F, G) \)); and (2) the iteration condition is associated with a subtask of its method [e.g., \( (F, \beta_i G) \)]. We call the relationship between task \( F \) and \( B \) a global interaction (denoted using a dotted line), because the interaction is between tasks at different levels. This kind of relationship can be expressed using a follow operator [e.g., \( (F, B) \).]
Figure 2. Dynamic behaviors in a task structure.

A TSE represents a collection of possible sequences of task invocations, referred to as task sequences. More formally, the semantics of a TSE over $\mathcal{T}$ is defined by a mapping to $\mathcal{S}$, which is the set of all possible task sequences for tasks in $\mathcal{T}$. The convention of viewing a conditional test as a task is also adopted, denoted as $\beta^?$. $\beta^?$ is a special control flow task that is invoked only when $\beta$ is tested to be true. Similarly, $\neg\beta^?$ is another special control flow task that is invoked only when $\neg\beta$ is tested to be true. The mapping function between a TSE and its corresponding task sequences is defined as follows:

**Definition 4.** Let $\mathcal{E}_1, \mathcal{E}_2$ be TSEs over $\mathcal{T}$, $T \in \mathcal{T}$ and $\beta$ be a well-formed formula. $\Gamma$ is a function that maps a TSE to a subset of $\mathcal{S}$ such that

1. $\Gamma(\emptyset) = \{\Box\}$, where $\Box$ denotes the empty task sequence.
2. $\Gamma(T) = \{t\}$, where $t$ denotes an invocation of $T$ and $\{\}$ denotes a task sequence.
3. $\Gamma(\mathcal{E}_1, \mathcal{E}_2) = \{s_1 \ast s_2\} \in \Gamma(\mathcal{E}_1) \land s_2 \in \Gamma(\mathcal{E}_2)$, where $\ast$ stands for concatenation.
4. $\Gamma([\mathcal{E}_1]) = \{\Box\} \cup \Gamma([\mathcal{E}_1])$. 
5. \( \Gamma(\beta, e_1 \lor \beta_2 e_2) = \Gamma(\beta_1, e_1) \cup \Gamma(\beta_2, e_2) \).
6. \( \Gamma((\beta e_1)^s) = \Gamma((\beta^? e_1)^s, (\neg \beta^? e)) \)  
   \( = \{ s * \neg \beta^? e \in \Gamma((\beta^? e)^s) \} \),  
   where \( \Gamma((\beta^? e)^s) = \bigcup_{i=1}^n \Gamma((\beta^? e_1)^s) \), and \( \Gamma((\beta^? e_1)) = \{ s_1 * s_2 * \ldots * s_i | s_j \in \Gamma(\beta^?, e_1) \} \),  
7. \( \Gamma(e_1; e_2) = \{ s_1 * s_3 * s_2 | s_1 \in \Gamma(e_1) \land s_2 \in \Gamma(e_2) \land s_3 \in \gamma \} \).

A TSE can be described at either the task level (a global TSE) or at the method level (a local TSE). In Figure 2, \( e_F = (F; B) \) and \( e_M = (\beta_1 D \lor \beta_2 E) \) are examples of global and local TSEs, respectively. A TSE at the task level is used to document global interactions between the current task and tasks at different levels that require special attentions in implementing and testing the system. A TSE at the method level documents the local control flow between subtasks of the method. To specify a global interaction, a global TSE often uses the “follow” operator (‘‘;”) to describe a set of partial task sequences. In contrast, a local TSE, which typically does not contain the “follow” operator, completely describes the control flow among the method’s subtasks. A TSE that does not contain the “follow” operator is called a complete TSE; otherwise, it is called a partial TSE.

There is another important difference between the semantics of the global TSE and the local TSE. A global TSE describes expected system behavior when the associated task is invoked. This property plays an important role in combining TSE’s at different levels. The current task to which a global TSE is associated is denoted by ‘‘1’. The semantics of the global TSE implies that it is meaningless to use optional, conditional, and selection operators for a task to which the task level TSE is associated, and therefore TSEs such as \( [1] \), \( (\beta^+ \lor T_i) \) are invalid TSEs at the task level.

C. Composition Operators

Multiple TSEs often need to be combined to obtain a global view of the system’s behavior from pieces of local behavior specifications. To achieve this, a composition operator is developed for combining two TSEs with a designated task \( T \) (called task-based composition operator), denoted as \( \oplus_T \). The semantics of composing two TSEs is defined through the corresponding set of task sequences. To compose two task sequences on a designated task invocation, denoted as \( (s_1 \oplus_T s_2) \), is defined by examining the relationship between the two sequences. We have identified three types of relationship between any two task sequences.

(a) Two task sequences may share a common subsequence containing an invocation of a designated task \( T \) [see Fig. 3(a)].
(b) Two task sequences are irrelevant w.r.t. a designated task \( T \), if at least one task sequence does not contain an invocation of \( T \) [see Fig. 3(b)].
(c) Two task sequences, \( s_1 \) and \( s_2 \), are said to have a conflicting subsequence w.r.t. a designated task \( T \), if (1) the task sequences contain the invocation of \( T \), (2) there exists a predecessor (or successor) of \( T \) in \( s_1 \) that is different from that in \( s_2 \) (which is called a conflicting predecessor or successor), and (3) the conflicting
predecessor (or successor) is not mapped to by a “follow” operator. Note that condition (3) is necessary in that, based on the definition of the follow operator, we can always find a conflicting predecessor (or successor) when composing a complete TSE with a partial one [see Fig. 3(c)].

**Definition 5 (Composing Two Task Sequences).** Suppose \( s_1 \) and \( s_2 \) are two task sequences. The composition of \( s_1 \) and \( s_2 \) on an invocation of a designated task \( T \), denoted as \( (s_1 \oplus_T s_2) \), is defined as follows:

- If \( s_1 \) and \( s_2 \) are sharing a common subsequence on an invocation of \( T \), then
  
  if \( s_1 = s'_1 \ast \alpha \) and \( s_2 = \alpha \ast s'_2 \) then \( s_1 \oplus_T s_2 = s'_1 \ast \alpha \ast s'_2 \)

  else if \( s_1 = s'_1 \ast \alpha \ast s''_1 \) and \( s_2 = \alpha \) then \( s_1 \oplus_T s_2 = s'_1 \ast \alpha \ast s''_2 \)

- If \( s_1 \) and \( s_2 \) are irrelevant, then \( s_1 \oplus_T s_2 = s_1 \cup s_2 \).

- If \( s_1 \) and \( s_2 \) have a conflicting subsequence containing an invocation of \( T \), then \( s_1 \oplus_T s_2 = \Box \).

The semantics of composing two TSEs conditioned on a designated task \( T \), is defined through its corresponding set of task sequences, \( \Gamma(\phi_1 \oplus_T \phi_2) \). That is, we can take a task sequence from \( \phi_1 \) and compose it with all task sequences in
by taking the union, and repeat the same process until all task sequences in $E_1$ are considered. This is formally defined below.

**Definition 6 (Composing Two TSEs).** Let $E_1$ and $E_2$ be two TSEs over $T$. Suppose $T$ is a task in $T$. The semantics of composing $E_1$ and $E_2$ conditioned on the designated task $T$, denoted as $E_1 \oplus_T E_2$, is defined as follows:

$$\Gamma(E_1 \oplus_T E_2) = \bigcup_{s_1 \in \Gamma(E_1), s_2 \in \Gamma(E_2)} (s_1 \oplus_T s_2)$$

It is trivial, from Definition 6, to show that the composition operator has the following properties:

- commutative: $E_1 \oplus_T E_2 = E_2 \oplus_T E_1$.
- associative: $(E_1 \oplus_T E_2) \oplus_T E_3 = E_1 \oplus_T (E_2 \oplus_T E_3)$.

However, not all TSEs in task structures are eligible for composition. There are two factors that affect the composability of TSEs: the conflicting task sequence and the follow operator. Intuitively, if there exists a conflicting task sequence in the to-be-composed TSEs, then we should not compose the TSEs. However, to compose a partial TSE with a complete TSE, we need to ensure that there does not exist a task sequence in the complete TSE that is conflicting with every possible task sequences of the partial TSE. A formal definition of composability is described below.

**Definition 7 (Composability).** Two TSEs, $E_1$ and $E_2$, are not composable conditioned on a designated task $T$, iff $\exists s_1 \in \Gamma(E_1), \forall s_2 \in \Gamma(E_2)$, such that $s_1$ and $s_2$ are conflicting w.r.t. $T$.

The following theorems for composing TSEs follow from Definition 4 and 6.

**Theorem 1.** Let $E$ and $E_T$ be two TSEs over $T$, and $T$ be a task in $T$. If $E$ does not contain any invocation of $T$ in $\Gamma(E_T)$, then $E \oplus_T E_T = E \lor E_T$.

**Theorem 2.** Let $E_1$, $E_2$, and $E_T$ be three TSEs over $T$. $E_T$ denotes a TSE in which $T$ appears. $(E_1 \lor E_2) \oplus_T E_T = (E_1 \oplus_T) \lor (E_2 \oplus_T E_T)$.

**Theorem 3.** Let $E_1$ and $E_2$ be two TSEs over $T$, $E_1 = (\alpha, T, \beta(\gamma_1, \gamma_2))$, and $E_2 = (T, \beta(\gamma_1; \omega))$, where $\alpha, \gamma_1, \gamma_2$ and $\omega$ are TSEs, and $\beta$ is a condition. $E_1 \oplus_T E_2 = (\alpha, T, \beta(\gamma_1, \gamma_2; \omega))$.

To summarize, there are two important characteristics of the task-based composition operator. First, to prevent any arbitrary composition, the composition is conditioned on a designated task to focus the attention on tasks that are

†All proofs of theorems can be found in Appendix A.
related to or interacted with the designated task. Second, combining a complete TSE with a partial one will help to fill in some of the missing details by fusing multiple pieces of behavioral specification.

As an illustration of the composition of TSEs, let us consider the following example. Suppose $E_M = (T_1, [T_3])$, and $E_T = (+; T_4)$.

$$
E_M \ominus_{T_3} E_T = ((T_1) \lor (T_1; T_3)) \ominus_{T_3} (T_3; T_4) \\
= (((T_1) \ominus_{T_3} (T_3; T_4)) \lor ((T_1, T_3) \ominus_{T_3} (T_3; T_4))) \\
= ((T_1) \lor (T_3; T_4) \lor (T_1, T_3; T_4))
$$

In this example, the global TSE is associated with $T_3$. However, $T_3$ is an optional task in its parent method. To compose $E_M$ and $E_T$, from Definition 4, we can transform $E_M$ into an equivalent TSE that has two components: one that always invokes $T_3$, the other that does not invoke $T_3$ at all. Based on Theorem 2, we can derive Eq. (2). Equation (3) follows directly from Theorem 1 and Definition 6.

Templates for tasks and methods specifications are shown in Figure 4. A task can be described using the triple $\langle \text{precondition}, \text{protection}, \text{postcondition} \rangle$, and the relationship between the task and tasks at different levels is captured by a global TSE. A task in a task structure can be refined into a set of methods that can accomplish the task. Each method can be further refined by specifying subtasks involved and sequencing relationship between subtasks using a local TSE. The specifications of tasks and methods are defined below.

**Definition 8 (Task Specification).** A specification of a task $T$ is a tuple $\langle \epsilon_T, \epsilon_T \rangle$, where $\epsilon_T$ is a triple $\langle B_T, P_T, R_T \rangle$, and $\epsilon_T$ is a task state expression involving tasks which have global interaction with $T$.

**Definition 9 (Method Specification).** A method specification of a task $T$ is a quadruple $\langle T, g, \mathcal{S}, E_M \rangle$, where $g$ is a formula describing the guard condition under which the method is applicable, $\mathcal{S}$ is a collection of subtasks, and $E_M$ is a TSE involving subtasks in $\mathcal{S}$.
In software specifications, it is desirable to compute the description of a state after the application of a transition. Similarly, researchers in AI planning are concerned about synthesizing a plan (that is, a sequence of actions) which involves constructing a state description after executing an action. Both software specifications and AI planning share a similar problem: How to keep track of what has not been changed? (This is the so-called frame problem.) The frame problem and its partial solutions are introduced in the next section. Section B elaborates on the proposed approach to the frame problem in the context of task structures.

A. Frame Problem in Software Specifications

In AI planning, frame axioms have been used to describe what has not been changed after the performance of an action. However, the problem arises when the number of frame axioms increases rapidly with the complexity of the application. Several partial solutions have been proposed. Fikes and Nilsson used the notion of differences, as well as ideas from the situation calculus to make the assumption that the initial world model would only be changed by a set of additions to, and deletions from, the statements modeling the world state—everything else remained unchanged. This assumption is sometimes called the STRIPS assumption. Hayes proposed using causal connections as a frame rule to the frame problem. The assumption is that if a term (or a nonlogical symbol) is not connected to an action, then any change to the action does not affect the term.

In software specifications, it is sometimes necessary to express explicitly the fact that certain variables do not change value (or state) due to the performance of a task. ASLAN provides four procedural operators: NoChange, ALTernate, Conditional, and Become, to explicitly specify unmentioned variables which are assumed not to have changed; meanwhile, the notation $\Theta$ in Z has been used to denote that any variable followed by the $\Theta$ will not change value, and $\Xi$ is used to indicate that any state space followed by $\Xi$ will not change. Four types of solutions to the frame problem are identified below:

- Domain-independent approaches: the solution is designed to be applicable to a wide variety of domains, for example, STRIPS assumption and the causal connection frame rule.
- Domain-independent, if-needed-basis approaches: a frame axiom is generated by the system on an if-needed-basis. A typical example of this approach is the NoChange operator in ASLAN, which is generated if the correctness of a specification cannot be proved due to the missing of values for variables.
- Application-specific approaches: frame axioms are tightly related to the application, and therefore, it is necessary for users to explicitly specify what will remain unaltered, for instance, the notion of invariant in software specifications such as $Z$.
- Operator-specific approaches: the solution is tied to a specific implementation of $\Xi$.

A similar classification scheme can be found in Ref. 23.
a specification language, and is only applicable to that particular language. The procedural operators in ASLAN are examples of this category.

To compute the description of a state involves not only the propagation of what has not been changed, but also the progression of what has been changed. Rosenschein’s progression operator and Waldinger’s regression operator are two similar approaches to only specifying what has been changed using a tuple \((\text{precondition}, \text{postcondition})\) in the state before and after an action. A progression operator is a function that maps conditions that are true immediately before a task (called before-state description) to conditions that are true immediately after the task (called after-state description), which is formally defined below.

**DEFINITION 10 (Progression Operator).** A progression operator, denoted as \(/;\), for \(T\) is a function mapping from formulas to formulas, such that for every triple \((b, d, a)\) \(\in T\), if \(\varphi\) is true in state \(b\), then \(\varphi / T\) is true in state \(a\).

\[
\forall (b, d, a) \in T, b \models \varphi \Rightarrow a \models \varphi / T
\]

In the proposed framework, for the purpose of verification, it is desirable to compute the description of a state that is obtained through a TSE to an initial state description, which is addressed in the next section.

**B. Computing State Descriptions**

As was noted by Borgida et al. that the frame problem is still an open issue and that it is likely no single general solution exists. In this section, the notions of progression operator, default rule, and frame axioms are introduced to address this problem in the context of task structures.

To extend the progression operator in the context of task structures, we first assume the description of a state is in a disjunctive normal form. An assumption about the forms of the precondition and the postcondition of a task is also made, that is, the precondition is a conjunction of literals; while the postcondition assumes a disjunctive normal form.

Second, the notion of default rule proposed by Reiter and frame axioms advocated by Hayes are incorporated into the semantics of the progression operator to specify state descriptions that remain unchanged after performing a task. The default rule and frame axioms are used to describe state descriptions that remain unaltered after performing a task. In default logic, unless there is knowledge to the contrary, we assume what is assumed to be true in the before state is also assumed to be true in the after state. Thus, the default rule is formulated as: a literal in the description of a state will remain intact if the literal is not contradictory to the postcondition of the task. This is formally defined below.

**DEFINITION 11.** Let the state description \(\varphi_i\) to a task \(T\) in a specification be in a disjunctive normal form: \(\varphi = (\varphi_1 \lor \varphi_2 \lor \cdots \lor \varphi_n)\), where \(\varphi_i\) in \(\varphi\) is a conjunction of literals: \(\varphi_i = (l_1 \land l_2 \land \cdots \land l_m)\), and \(\varphi_j \rightarrow \mathcal{R}_T\). \(\varphi_i / T = \mathcal{R}_T \land \varphi_i\), where \(\varphi_j\) is a conjunction of literals and \(l_i\) in \(\varphi_i\) such that \(l_i\) is not contradictory to \(\mathcal{R}_T\).
Several features in the proposed framework facilitate the formulation of frame axioms: domain model, state model, and protections. Both the value of a variable and the state of an object in the model specification of a task can be changed after performing the task, which is analogous to the value of a variable (or a state variable) in conventional software specifications. Most software specifications methods use a variable with a prime (that is, ’) to represent the old value of the variable and a variable without the prime for new value of the variable, which is a convention adopted in the proposed framework. The same approach is used for representing the change of status for a state variable in conventional software specifications. However, in the proposed framework, the notion of state objects in the state model, represented using either a proposition or a predicate, is used to reflect the change of status for an object. Therefore, predicates in the domain model of a task should never be changed after the task. From the definition of protections in the functional specification of a task, the predicates in the protection should remain intact after the execution of the task. Based on the above discussion, the following lemma can be derived.

**Lemma 1.** Let $D$ be the domain model of a specification and the state description $\varphi_i$ to a task $T$ in the specification be in a disjunctive normal form: $\varphi = (\varphi_1 \lor \varphi_2 \lor \ldots \lor \varphi_n)$, where $\varphi_i$ in $\varphi$ is a conjunction of literals: $\varphi_i = (l_1 \land l_2 \land \ldots \land l_n)$, and $\varphi_i \rightarrow R_T^T, \varphi_i/T = R_T^T \land \varphi$, where $\varphi_i$ is a conjunction of literals and $l_i$ in $\varphi_i$ in $\varphi$ such that $l_i$ satisfies any of the following:

1. $l_i$ is a domain model predicate.
2. $l_i$ is a predicate in the protection of $T$.

Third, since the proposed framework adopts the notion of task structures for functional decomposition, **the progression is only performed within each task** (that is, the task, its methods, subtasks of the methods, and tasks related through global interactions). In another word, the initial state description is the conjunct of the precondition of the task and the guard condition of the task’s method, and the state descriptions are progressed through the composed TSE of the task’s method TSE and the TSE of any high level task that has global interactions with any task in the method.§ For example, in Figure 2, to propagate through the task $D$ is to progress through the composed TSE: $\varepsilon_{M_1} \oplus_{B} \varepsilon_{B}$ with $B_D \land \beta_{M_1}$ as the initial state description.

To progress a state description through an immediate follow operator (“,”) in a TSE, there are two conditions which need to be satisfied: (1) the state description is in a disjunctive normal form, and (2) the precondition of a task $T$ that immediately follows the operator can be derived from every disjunct in the state description. The result of the progression through the immediately follow operator will be equivalent to the disjunction of applying the progression operator

§To make it consistent in the presentation, “/” is used to represent both the progression operator through a task and a TSE.
to each individual disjunct over the task $T$. This is formally described using the following theorem.

**Theorem 4.** Suppose the initial state description $\varphi$ to a TSE $\mathcal{E}$ is in a disjunctive normal form: $\varphi = (\varphi_1 \lor \varphi_2 \lor \ldots \lor \varphi_n)$, where $\varphi_i$ in $\varphi$ is a conjunction of literals. If $\mathcal{E} = (e, T)$ and $\forall i \varphi_i \rightarrow B_T$, then $(\varphi_1 \lor \ldots \lor \varphi_n)/\mathcal{E} = \varphi_1/T \lor \ldots \lor \varphi_n/T$.

To progress a state description through a selection operator (“$\lor$”) in a TSE, several conditions need to be satisfied: (1) the state description is in a disjunctive normal form, (2) at least one disjunct in the state description can deduce the precondition of the task $T$ (i.e., $B_T$) that is used with the selection operator. The result of the progression is a disjunction of applying the progression operator over $T$ to those disjuncts that can derive $B_T$, and applying the progression operator to those disjunctions that cannot derive $B_T$.

**Theorem 5.** Suppose the initial state description $\varphi$ to a TSE $\mathcal{E}$ is in a disjunctive normal form: $\varphi = (\varphi_1 \lor \varphi_2 \lor \ldots \lor \varphi_n)$, where $\varphi_i$ in $\varphi$ is a conjunction of literals. $\mathcal{E} = (T_1 \lor T_2)$

If $\varphi = \varphi' \lor \varphi''$ where $\varphi'$ and $\varphi''$ are in disjunctive normal form, such that $\forall \varphi_i in \varphi'$, $\varphi_i \rightarrow B_{T_1}$ and $\forall \varphi_i in \varphi''$, $\varphi_i \rightarrow B_{T_2}$, then $(\varphi' \lor \varphi'')/\mathcal{E} = \varphi'/T_1 \lor (\varphi'')/T_2$.

Based on the above discussion, the following algorithm will compute the state description through a TSE $\mathcal{E}$ to an initial state description $\varphi$. A specification is said to be weakly inconsistent if a consistency condition cannot be proved; meanwhile, if a consistency condition can be disproved, then the specification is said to be strongly inconsistent (the detail of levels of inconsistency is described in the next section).

**Algorithm 1 (Progression Algorithm for a TSE).** Assume that the initial state description $\varphi$ to a TSE $\mathcal{E}$ is in disjunctive normal form: $\varphi = (\varphi_1 \lor \varphi_2 \lor \ldots \lor \varphi_n)$, where $\varphi_i$ in $\varphi$ is a conjunction of literals. Domain model predicates are denoted as $\varphi_i$.

1. $\mathcal{E} = (e, T)$
   - Case 1: If $\forall i \varphi_i \rightarrow B_T$, then $(\varphi_1 \lor \ldots \lor \varphi_n)/\mathcal{E} = \varphi_1/T \lor \ldots \lor \varphi_n/T$.
   - Case 2: If $\exists \varphi_i$ such that $\varphi_i \rightarrow B_T$, then weakly inconsistent.
   - Case 3: If $\exists \varphi_i$ such that $\varphi_i \rightarrow \neg B_T$, then strongly inconsistent.

2. $\mathcal{E} = (T_1 \lor T_2)$
   - Case 1: If $\varphi = \varphi' \lor \varphi''$ where $\varphi'$ and $\varphi''$ are in disjunctive normal form, such that $\forall \varphi_i in \varphi'$, $\varphi_i \rightarrow B_{T_1}$, and $\forall \varphi_i in \varphi''$, $\varphi_i \rightarrow B_{T_2}$, then $(\varphi' \lor \varphi'')/\mathcal{E} = \varphi'/T_1 \lor \varphi''/T_2$.
   - Case 2: If $\exists \varphi_i in \varphi$ such that $\varphi_i \rightarrow B_{T_1}$ or $\varphi_i \rightarrow B_{T_2}$, then weakly inconsistent.
   - Case 3: If $\exists \varphi_i in \varphi$ such that $\varphi_i \rightarrow \neg B_{T_1}$ and $\varphi_i \rightarrow B_{T_2}$, then strongly inconsistent.

3. $\mathcal{E} = (e; T)$
\[ \varphi_1 \varepsilon = R_T \land \varphi_0 \land P_T \land \varphi', \text{ where } \varphi' \text{ is the state description constructed by the expression performed immediately before } T. \]

\[ \varepsilon = (\beta T)^* \]

\[ \varphi_1 \varepsilon = -\beta \land \varphi_0 \land P_T \land \varphi, \text{ where } \varphi' \text{ is the state description propagated immediately before exiting the iteration.} \]

**V. VERIFICATION OF SPECIFICATIONS**

The verification of a specification is important because it helps early detection of errors in the software life cycle to produce an adequate specification, thus eliminating possible design failures and reducing product costs. The proposed framework provides two levels of inconsistency checking in a single layer: strong and weak inconsistency. A system specification is weakly inconsistent if its consistency cannot be proved. On the other hand, a system specification is strongly inconsistent if its consistency can be disproved. The proposed framework also checks for if a refinement is consistent by comparing the specificity of conditions at the high layer to that of the low layer. Types of faults that could result in an inconsistent process specification include protection violation, precondition violation, postcondition violation, and incorrect TSEs. Missing methods will result in incompleteness in the specification.

A specification of a task or a method of a system is consistent if (1) the precondition of a task in the TSE of a task or a method can be deduced from the state description before performing the task; (2) the postcondition of the task can be deduced from the description of the state after performing the task; and (3) the protection of the task can be deduced from state descriptions before, after, and during performing the task.

Thus, a method specification of a task is weakly inconsistent if (1) the precondition of a task in the TSE of the method cannot always be deduced to be true from the state description after the task; (2) the postcondition of the task cannot always be deduced to be true from the state description before the task; and (3) the protection of the task cannot always be deduced to be true from the state description before, after, or during the task. In contrast, a method specification of a task is strongly inconsistent if (1) the precondition of a task in the TSE of the method can sometimes be proved to be false from the state description after the task; (2) the postcondition of the task can sometimes be proved to be false from the state description before the task; and (3) the protection of the task can sometimes be proved to be false from the state description before, after, or during the task.

The consistent refinement checking is performed between adjacent levels of a task and its methods. The mechanism for verifying the consistency is to check for if a refinement is consistent by comparing the specificity of conditions at the high level to that of the low one.

**Definition 12 (Specificity of Conditions).** A condition \( C_1 \) is exclusively more specific, denoted as \( < \), than condition \( C_2 \) if and only if for all states \( s \), if \( s \) satisfies \( C_1 \), then \( s \) satisfies \( C_2 \), but not vice versa (i.e., they are not equivalent).
\[ C_1 < C_2 \iff (\forall s, s \models C_1 \Rightarrow s \models C_2) \land (\forall s, s \models C_2 \Rightarrow s \models C_1) \]

**DEFINITION 13 (Consistent Refinement).** A refinement of a task \( T \) is consistent with \( T \)'s specification, if and only if the collection of state transition \( \langle b, d, a \rangle \) of \( T \)'s methods is a subset of the collection of \( T \)'s state transition \( \langle b, d, a \rangle \in T \).

**THEOREM 6 (Inconsistent Refinement).** Let \( \langle C_T, \varepsilon_T \rangle \) be a task specification of \( T \), and \( \langle T, g, \Delta T, \varepsilon_M \rangle \) a method specification \( M \) of \( T \). \( M \) is an inconsistent refinement of \( T \), if one of the following holds:

1. \( B_T \not\models B_M \).
2. \( P_T \not\models P_M \).
3. \( R_T \not\models R_M \).
4. \( R_T \land I_T \not\models R_M \land I_M \).

**THEOREM 7 (Protection Violation).** A refinement of \( T \) is inconsistent, if \( \varphi \) is protected during task \( T \), there exists a subtask \( T' \) of a method \( M \) in \( T \)'s refinement such that the rigid postconditions of \( T' \) can deduce \( \neg \varphi \).

**DEFINITION 14 (Complete Refinement).** A refinement of a task \( T \) is complete with \( T \)'s specification, if and only if the collection of the before states of \( T \)'s method specifications is equivalent to the collection of the before states of \( T \).

The verification of the process specification starts from the highest level task, and focuses one task at a time.

**ALGORITHM 2.** To verify a task, there are two major steps involved:

1. The specification of a task \( T \) is compared with the method that contains \( T \) (called \( T \)'s parent method) and other subtasks in the method to prove or disprove \( T \)'s precondition using the resolution algorithm.
2. The specification of a task \( T \) is compared with those methods (and their subtasks) that accomplish \( T \) (i.e., \( T \)'s child methods) for consistency and completeness checking.

(a) The specification of a \( T \)'s method is **consistent** with \( T \) if
   i. the description of the state prior to executing the method, referred to as the before state description, is semantically more specific than \( T \)'s precondition,
   ii. the description of the state after executing the method, referred to as the after state description, is more specific than \( T \)'s rigid postcondition,
   iii. the protection of \( T \) is not violated by any subtasks of \( T \).

(b) The specification of \( T \)'s methods is considered **complete** if the union of their before state descriptions is equivalent to \( T \)'s precondition.
VI. AN EXAMPLE

Based on an original implementation and documentation of R1/SOAR, the reimplementation of a major portion of the well-known computer configuration expert system R1,\textsuperscript{11} we have constructed, in a reverse engineering fashion, a specification for R1/SOAR to evaluate the proposed framework.

R1/SOAR focuses on the unibus configuration task of the entire configuration task of R1. The problem is given a customer's order, place the modules on the order into backplanes, and the backplanes into boxes on the correct unibus. The backplanes need to be cabled together, and the panel space allocated when required by the modules. If the containers on order are not sufficient to hold the modules, additional containers must be ordered.

The following example is from a subset of the task-based specifications of R1/SOAR. Two major tasks of concern in this example are configure modules and configure a module. The functionality of the task configure modules is to configure as many modules as possible into the current backplane while maintaining the optimal ordering. This is achieved by iterating between two subtasks: (1) configure a module, which attempts to configure the current module into the current backplane, and (2) obtain the next module, which gets an ordered module that follows the current module in the optimal ordering.

One of the errors that a knowledge engineer is likely to make is about the situation when the current module fails to be configured into the current backplane. A correct specification should indicate that obtain the next module should be skipped under the situation, since we have not finished configuring the current module yet. It is also desirable to indicate that if the current module cannot be configured into the current backplane, the system should later find a backplane suitable for the current module before configuring the next module. The specifications for the tasks and methods relevant to this example are shown in Figures 5, 6, and 7. We discuss the state model, functional specification, and behavioral specification relevant to the example below, and show how the common error mentioned above can be detected by the verification algorithm.

Functional specification: An example of protections in R1/SOAR is that the optimal ordering in which modules are configured should be maintained in the
Method:
- Parent Task: Configure Modules
- Guard Condition: TRUE
- Subtasks:
  - $T_1$: configure a module.
  - $T_3$: obtain the next module.
- TSE: $(T_1, [T_3])^*$

**Figure 6.** A method of the task configure modules.

Figure 7. Configure a module.
Verification: If the TSE of the method were \((T_1, T_2)^*\), a strong inconsistency will be detected by the verification algorithm because the postcondition of configure a module can be formally expressed using the following logic formula:

\[
\text{configured}(\text{current} \rightarrow \text{module}) \\
\lor \text{incompatible}(\text{current} \rightarrow \text{module}, \text{current} \rightarrow \text{backplane})
\]

Invoking the sequence, \((\ldots T_2, T_1 \ldots)\), when the current module is incompatible with the current backplane leads to configuring the next module before configuring the current one. Hence, two strong inconsistencies can be detected: (1) the precondition of the task obtain the next module, which states that the current module should have been configured, can be violated, and (2) the protection that the optimal ordering of modules should be maintained can also be violated.

**VII. RELATED WORK**

We briefly compare our methodology with major alternative specification methodologies for knowledge-based systems below.

- Zualkernan and Tsai have developed two specification methodologies for expert systems: an object-oriented methodology, and a knowledge-model methodology.\(^6\) The object-oriented methodology first establishes the object class for specifying instances of each of the expert activity involved, which is followed by the definition of the methods for each class, and then the instantiation of class instances. The knowledge-model methodology describes the concept structures and the goal structures that are required to solve the problem. Both methodologies provide a top-down approach and have been demonstrated for a heuristic classification system. Verification and validation are performed through the interaction with experts. Compared to our approach, both methodologies offer two benefits: (1) the specification is executable, and (2) the methodologies are easier to be applied to the heuristic classification type expert systems.

- Slagle has used conceptual graphs for specifying a heuristic classification expert system.\(^4\) His approach begins with type definitions, identifies the high-level lines of reasoning focused on a few concepts, constructs implications, facts, type lattice, and canonical graphs. Like Zualkernan and Tsai’s work, Slagle’s methodology is suitable for heuristic classification type systems, whose knowledge can be specified by constructing an inference network that relates pieces of evidence to hypotheses.

- A problem space approach to expert system specification has been proposed by Yost and Newell.\(^27\) The approach is designed specifically for the SOAR architecture. It is difficult to see how the methodology can be applied to other AI architectures. Furthermore, verification of the specification has not been demonstrated.

The proposed approach offers two important advantages: (1) It provides a solid automatic verification capability. (2) It provides a detailed functional decomposition technique.

**VIII. CONCLUSION**

In this article, the formal foundation for task structures has been proposed to formulate two major features in process specification: state-transition and process refinement. The general form of state-transition model\(^ {24-28}\) is thus adopted
for the purpose of formulating and analyzing the notion of tasks in the structures. The notion of task structure (i.e., task/method/subtask) is formalized for the analysis of task process refinement. Task state expressions, based on regular expressions, are used to describe expected behavior specifications. Progression operators and frame axioms are adopted for constructing the description of a state through a TSE to an initial state description. Levels of inconsistency: strong and weak, have been identified. The formalism provided by the framework facilitates the verification of specifications. Verification for consistency and completeness is performed for pieces of the specification within one abstraction level or between multiple levels based upon their formal semantics.

Our approach offers several important benefits for specifying knowledge-based systems. First, the incorporation of task structures provides a detailed functional decomposition technique for organizing and refining functional and behavioral specifications of a knowledge-based system. Second, the use of TSEs can assist not only in documenting the expected control flow and module interactions, but also in verifying that the behavioral specification is consistent with the system’s functional specification. Third, the verification algorithm is based on its well-defined formal foundation, and is achieved by centering around one functional unit at a time. Finally, the distinction between soft and rigid conditions makes it possible to specify conflicting functional requirements.

Future research will consider the following directions: (1) to enhance the verification capability by an extension of the proposed framework which can be facilitated using a principled knowledge representation language (e.g., conceptual graphs); and (2) to perform a trade-offs analysis for informal requirements.

This research is partially supported by National Science Council (Taiwan, R.O.C.) under Grant NSC 86-2213-E-008-006. The author would like to thank Professor John Yen for his invaluable comments on an early draft of this article.

IX. APPENDIX: PROOFS

THEOREM 1. Let \( \mathcal{E} \) and \( \mathcal{E}_T \) be two TSEs over \( \mathcal{T} \), and \( T \) be a task in \( \mathcal{T} \). If \( \mathcal{E} \) does not contain any invocation of \( T \) in \( \Gamma(\mathcal{E}_T) \), then \( \mathcal{E} \bigoplus_T \mathcal{E}_T = \mathcal{E} \bigvee \mathcal{E}_T \).

Proof. Since \( \Gamma(\mathcal{E}) \) does not contain any task invocation of \( T \) in \( \Gamma(\mathcal{E}_T) \), it follows from Definition 6 that

\[
\Gamma(\mathcal{E} \bigoplus_T \mathcal{E}_T) = \Gamma(\mathcal{E}) \cup \Gamma(\mathcal{E}_T)
\]

From Definition 4, we have

\[
\Gamma(\mathcal{E} \bigvee \mathcal{E}_T) = \Gamma(\mathcal{E}) \cup \Gamma(\mathcal{E}_T)
\]

Combining Eqs. (6) and (7), we get \( \Gamma(\mathcal{E} \bigoplus_T \mathcal{E}_T) = \Gamma(\mathcal{E} \bigvee \mathcal{E}_T) \). Therefore, it is proved that

\[
\mathcal{E} \bigoplus_T \mathcal{E}_T = \mathcal{E} \bigvee \mathcal{E}_T \).

\[\square\]
THEOREM 2. Let $E_1$, $E_2$, and $E_T$ be three TSEs over $T$. $E_T$ denotes a TSE in which $T$ appears. $(E_1 \cup E_2) \oplus T E_T = (E_1 \oplus_T E_T) \cup (E_2 \oplus_T E_T)$.

Proof. The theorem can be trivially proved based on the Definition 6 and Definition 4. Based on Definition 6, we have

$$\Gamma((E_1 \oplus_T E_2)) = \bigcup_{\forall s_1 \in \Gamma(E_1), \forall s_2 \in \Gamma(E_2)} (s_1 \oplus s_2)$$

From Definition 4, we know that $\Gamma(E_1 \cup E_2) = \Gamma(E_1) \cup \Gamma(E_2)$. Therefore,

$$\Gamma(E_1 \cup E_2) \oplus T E_T = \Gamma(E_1 \oplus_T E_T) \cup \Gamma(E_2 \oplus_T E_T)$$

THEOREM 3. Let $E_1$ and $E_2$ be two TSEs over $T$. $E_1 = (\alpha, T, \beta(\gamma_1, \gamma_2))$, and $E_2 = (T, \beta(\gamma_1; \omega))$, where $\alpha$, $\gamma_1$, $\gamma_2$ and $\omega$ are TSEs, and $\beta$ is a condition. $E_1 \oplus_T E_2$ is a subsequence of $E_T$.

Proof. From Definition 4, we know that

$$\Gamma(\alpha, T, \beta(\gamma_1, \gamma_2)) = \{t \in T | t is an invocation of T and \beta is a task for testing T \land \beta \land \square \text{ is an empty task sequence}\}$$

and that (1) for any sequence $\langle t \cdot \beta \cdot s_2 \cdot s_3 \cdot s_4 \rangle \in \Gamma(T, \beta(\gamma_1; \omega))$, we know that there always exists a sequence $\langle t \cdot \beta \cdot s_2 \cdot s_3 \cdot s_4 \rangle$ such that $\langle t \cdot \beta \cdot s_2 \cdot s_3 \cdot s_4 \rangle$ is a subsequence of $\langle t \cdot \beta \cdot s_2 \cdot s_3 \cdot s_4 \rangle$ (i.e., $\langle t \cdot \beta \cdot s_2 \cdot s_3 \cdot s_4 \rangle = \langle t \cdot \beta \cdot s_2 \cdot s_3 \cdot s_4 \rangle$), where $s_2, s_3, s_4 \in T$.

Based on Definition 6, the result of composition is

$$\Gamma(E_1 \oplus_T E_2) = \{t \in T | t is an invocation of T and \beta is a testing task for T \land \beta \land \square \text{ is an empty task sequence}\}$$
THEOREM 4. Suppose the initial state description \( \varphi \) to a TSE \( \varepsilon \) is in a disjunctive normal form: \( \varphi = (\varphi_1 \lor \varphi_2 \lor \ldots \lor \varphi_n) \), where \( \varphi_i \) in \( \varphi \) is a conjunction of literals.

If \( \varepsilon = (e, T) \) and \( \forall \phi_i \in R_T, T \vdash (\varphi_1 \lor \ldots \lor \varphi_n) \lor (e \lor T) \), then \( (\varphi_1 \lor \ldots \lor \varphi_n)/\varepsilon = \varphi/T \lor \ldots \lor \varphi_i/T \).

Proof. The proof of this theorem is outlined below: Since \( b \models \varphi_1 \lor \ldots \lor \varphi_n \) and \( b \models \varphi_1 \lor \ldots \lor \varphi_n \), we can derive that \( a \models (\varphi_1 \lor \ldots \lor \varphi_n)/T \). Thus, the theorem is proved.

THEOREM 5. Suppose the initial state description \( \varphi \) to a TSE \( \varepsilon \) is in a disjunctive normal form: \( \varphi = (\varphi_1 \lor \varphi_2 \lor \ldots \lor \varphi_n) \), where \( \varphi_i \) in \( \varphi \) is a conjunction of literals.

If \( \varphi = \varphi' \lor \varphi' \), where \( \varphi' \) and \( \varphi'' \) are in disjunctive normal form, such that \( \forall \phi_i \in \varphi', T_1 \vdash (\varphi_1 \lor \varphi''/T_1) \) and \( \forall \phi_j \in \varphi''/T_2 \), then \( (\varphi' \lor \varphi'')/\varepsilon = \varphi'/T_1 \lor (\varphi''/T_2) \).

Proof. This theorem can be trivially proved by showing that since \( \varphi'' \) can not deduce the precondition of \( T \), \( \varphi''/T_1 \) is not defined. Similarly, \( \varphi''/T_2 \) is not defined either.

THEOREM 6. (Inconsistent Refinement). Let \( \langle C_T, \varepsilon_T \rangle \) be a task specification of \( T \), and \( \langle T, g, \mathcal{R}_T, \mathcal{R}_M \rangle \) a method specification \( M \) of \( T \). \( M \) is an inconsistent refinement of \( T \), one of the following holds:

1. \( \mathcal{R}_T \not\models \mathcal{R}_M \).
2. \( \mathcal{R}_T \not\models \mathcal{R}_M \).
3. \( \mathcal{R}_T \not\models \mathcal{R}_M \).
4. \( \mathcal{R}_T \not\models \mathcal{R}_M \).

Proof. This theorem can be trivially proved by following directly from Definition 12 and 13.
THEOREM 7. A refinement of $T$ is inconsistent, if $\varphi$ is protected during task $T$, there exists a subtask $T'$ of a method $M$ in $T$'s refinement such that the rigid postconditions of $T'$ can deduce $\neg\varphi$.

Proof. Assume that the refinement is consistent and $\neg\varphi$ is true after the refinement. From Definition 2, protection has to be held for after state of $T$. Since the refinement is consistent, we know that $\mathcal{P}_T \supset \mathcal{P}_M$, which implies that $\varphi$ is true in $\mathcal{P}_M$. Thus, the theorem is proved by contradiction.

References


