A Temporal Framework for Assembly Sequence Representation and Analysis

Kiam Tian Seow and R. Devanathan, *Senior Member, IEEE*

Abstract— **A unifying temporal logic framework is proposed for modelling, specifying, and analyzing mechanical assembly sequences. Propositional temporal logic of Manna and Pneuli is the assertion language used for representing and analyzing all constraints or conditions that assembly sequences must satisfy. The generalization of the existing representations through the proposed framework is demonstrated. A new concept of reverse-equivalence, relating assembly and disassembly forms of knowledge, is introduced. It is shown that existing and new assembly sequence properties can be formulated and rigorously proved via *mechanical theorem proving* based on two assembly process-axioms, and the language inference rules and theorems. Comparison of the proposed framework with the existing representation schemes highlights the main strengths of the proposed framework, viz., temporal expressiveness, strong mechanical manipulation capability and precise formalism. The implementation of the framework on an *IPC SUN SPARC* workstation using *Quintus Prolog* for automatic generation of assembly sequences is described. A simple example illustrates the use of the proposed framework for representation, evaluation and selection of feasible assembly sequences.**

I. Introduction

ASSEMBLY PLANNING plays a fundamental role in the manufacture of most mechanically assembled products. In this paper, we are concerned with the assembly sequence representation issue. An implicit assembly sequence representation refers to the conditions or constraints that assembly sequences must satisfy. With a growing need to systematize and computerize the generation of correct and complete assembly sequences [1], it has become increasingly necessary to address the related problem of completeness and manipulability of representation. There is a need to develop a formal framework that can completely and compactly represent the assembly sequences, and provide an inference mechanism to efficiently facilitate the analysis of complex precedence relationships which are invariably involved.

One such development is Fox's language [2] for implicit representation of serial robotic tasks. Implicit representations have also been formulated as "establishment conditions" (first introduced by Bourjault [3]) and "precedence relationships" by Homem de Mello and Sanderson [4], "liaison precedence relationships" by De Fazio and Whitney [5], "MP() and NL()" predicates by Huang and Lee [6] and "assembly constraint" by Ayoub and Doty [7]. Except for the "establishment conditions" scheme, the other schemes are quite similar, differing only in the degree of formal treatment and the notation of the precedence symbols used. In general, as shown later, these schemes suffer from the lack of expressiveness to assert a wider vocabulary of precedence ordering constraints, as well as an adequate inference mechanism to facilitate concise and insightful analysis over the representations.

The feasibility of assembly sequences depends not only on the geometric or inherent ordering constraints [8] (also called hard constraints[1] in Huang and Lee [6]), but also on the soft-constraints (also called planner directives in Ayoub and Doty [7]) that specify additional precedence requirements based on non-geometric criteria. The nature of the soft constraint is such that it can be used in decision making by the designer to narrow down the choice to a few "good" sequences from among the possible assembly sequences [9]. The designer may choose to apply the soft constraints along with the hard constraints before generating the assembly sequences, or he may defer his decision to apply the soft constraints until after all the assembly sequences are generated based only on the hard constraints.

Existing techniques to find the appropriate sequences include an algorithmic search over a graphical representation such as the AND/OR graph [10], using admissible heuristics of some evaluation criteria [11]. However, such an approach may not provide sufficient mechanical manipulability and not much insight can be gained from the selected sequences. In Baldwin *et al.* [9], [12], evaluation and selection are done by manual editing via computer-guided instructions and criteria options. In Henrioud and Bourjault [13], the assembly constraints are classified and formalized. In Delchambre [14], all temporal constraints are specified by a single predicate *precedence-order()* only.

A temporal logic framework unifying all the implicit representations, upon which they could be precisely described and therefore mathematically reasoned upon, so as to generate the desired assembly sequences, is proposed in this paper. We present a temporal logic formulation of assembly sequence properties that forms the basis for analyzing assembly sequences. Propositional temporal logic of Manna and Pneuli [15], [16] is adapted for the purpose. The proposed framework standardizes the representation of the elements of precedence knowledge that could model the hard constraints. Table I shows the various symbolic representations, as used by previous researchers, that correspond with those asserted

Manuscript received December 11, 1992; revised July 29, 1993.

The authors are with the School of Electrical and Electronic Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 2263, Republic of Singapore.

IEEE Log Number 9215780.

[1] In our work, the term *hard-constraint* is used for precedence constraint due to part geometry. It is strictly essential and invariant for a given assembly, hence the term "*hard*".

TABLE I
VARIOUS PROPOSED REPRESENTATIONS OF ASSEMBLY TASK-PRECEDENCE

No.	Operator name	Symbolic representation as used in				
		Proposed framework	Huang [6]	Homem de Mello [4]	Ayoub [7], Fox [2]	De Fazio [5]
1	Precede	$p_1 \mathcal{P} p_2$	MP(p_1,p_2)	$p_1 < p_2$	$p_1 < p_2$	$p_1 \Rightarrow p_2$[1]
2	Until	$p_2 \mathcal{U} p_1$	NL(p_1,p_2)	$p_1 \leq p_2$	—	—

[1]We use the symbol "$\Rightarrow$" in place of the symbol "$\rightarrow$" used in [5] so as to avoid conflict with the logical implications connective used from Section II-B onward.

by two temporal operators. Except for Huang and Lee [6], no apparent attempt is made to formally derive the assembly sequence properties. This is largely due to the lack of mathematical support for mechanical manipulation of these knowledge elements. The use of temporal logic can not only extend the knowledge forms beyond these two elements, but also can provide the inference mechanism for mechanical derivation of many useful properties, including those provided by Huang and Lee [6]. By virtue of its wider set of temporal operators and established rules and theorems providing the strong manipulability, it could possibly provide a flexible framework by which design constraints (both hard and soft-constraints) could be specified and logically integrated to obtain the desired or feasible set of precedence specifications.

It should be pointed out, however, that it appears difficult to specify a correct and complete set of hard constraints for complex assemblies. This difficulty is regardless of the form of representation used including our proposed framework. Essentially, there have been two broad approaches to the generation of hard constraints. The approach due to Bourjault [3] and De Fazio and Whitney [5] lends itself to interactive systems. The second approach is to generate the hard constraints through an algorithm operating on a graphical model of the product assembly [6], [17]. However, the physical and geometric reasoning required for determining the constraints appears quite complex and are themselves active topics of further research [17].

The paper is organized as follows: Section II presents the necessary background for the use of temporal logic as an assembly constraint language. Section III presents a temporal formulation of the sequence behavior. Section IV discusses the general framework for assembly modelling, specification and analysis. Section V presents the software implementation for automatic assembly sequence generation and verification. Section VI provides an illustrative example. Section VII compares and discusses the proposed framework with existing representations. Section VIII summarizes the paper.

II. BACKGROUND

A. Assembly

In our formulation, the assembly process for mechanical assemblies is assumed to consist of well-defined tasks all of which when executed, will result in the assembled product. Each task is assumed to consist of a set of operations in which one or more of the connections (or liaisons) between parts of the assembly are established. The states of the assembly process are represented in terms of the status (done or not done) of the different tasks. Any transition between two distinct states corresponds to the carrying out of the operations of the task involved. The states are identified with the nodes and the transition between states by edges of a graph representing the assembly process. A given assembly sequence can then be identified in terms of a subset of nodes, put in a particular sequence, defining a path in the graph. Our view of the assembly plan is similar to that used by Huang and Lee [6] and De Fazio and Whitney [5].

B. MAPS and Temporal Logic

Linear time temporal logic is essentially a language for describing sequences of situations and for reasoning about them. In a purely temporal interpretation, a sequence of situations represents the evolution of the state of the world with time. By considering an interpretation where the sequence of situations represents the successive discrete states an assembly goes through to be assembled from its various parts, we could use temporal logic for assembly sequence representation and analysis.

Suppose there are exactly n tasks for a product assembly. Then the status of these tasks (that is, whether each task is done or not) at each stage where some task(s) just begin execution constitute an abstract state of the assembly or disassembly process. By viewing all such possible states, a total of 2^n of them, as the problem space, the process of assembly or disassembly becomes a process of state-transitions in this problem space. The knowledge about product assembly or disassembly becomes that of state-transitions. We will now formally introduce the problem space before defining some temporal operators that can be interpreted by the state-transitions in the next section.

A mechanical assembly planning structure (MAPS) for an assembly is the problem space that completely encompasses all feasible assembly sequences and can therefore be formalized as a 2-tuple model $\mathcal{M}$ given by

$$\mathcal{M} = (S, R), \text{ where}$$

1) S is a finite set of states of true atomic propositions of distinct assembly tasks $\{p_1, p_2, \cdots, p_n\}$ that constitute the whole assembly.
2) R is a binary relation on S ($R \subseteq S \times S$). which gives the possible transitions between states and must be total,[2] that is, $\forall x \in S \; \exists y \in S \mid (x, y) \in R$.

[2]Being "total" is required to ensure a complete state-transition space with no isolated states.

MAPS is a simple Kripke structure [18] or state-transition graph $\mathcal{M}$ of discrete states with only assembly task-symbols and without any global constants[3]; and any finite list of states or finite trajectory $\sigma \in \mathcal{M}$ is a possible assembly sequence.

Propositional temporal logic [16] is a language of propositional logic augmented with temporal operators to facilitate reasoning over the space of sequences of states. The logical "AND," "OR," "Negation," "Implication," and "Equivalence" connectives in classical logic are denoted by the symbols $\wedge$,$\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$ or $\equiv$ respectively. The sum ($\sum$) and product ($\prod$) symbols denote a string of logical "OR" and "AND" operations respectively. Temporal operators include Always $\Box$, Eventually $\diamond$, Next $\bigcirc$, Until $\mathcal{U}$ and Precede $\mathcal{P}$. A logic formula w containing at least one such operator is called a temporal formula.

An interpretation, in the context of assembly, refers to a possible assembly sequence of the defined problem space that "interprets" (that is, makes true or satisfies) the temporal formula describing some assembly precedence constraints. The suffix of a sequence refers to its partial sequence that begins from a given state.

Before defining the temporal operators in terms of the satisfaction relations, the formal notions of a general interpretation and its suffix are first given.

Definition 1 [Interpretation I]: An interpretation I is a 3-*tuple* ($\mathcal{M}$,$\mathcal{G}$, σ), *where* σ *is a temporal sequence of states in structure* $\mathcal{M}$ ($\sigma = s_0 - s_1 - s_2 - \cdots$) *and* $\mathcal{G}$[4] *assigns a value for every global constant in* $\mathcal{M}$.

With the above definition of Interpretation, the structure $\mathcal{M}$ can therefore be regarded as a restricted class of possible semantics or interpretations ($\mathcal{M}$, σ) of temporal logic *TL* [16] by which precedence and other state constraints $\mathcal{J} \in TL$ can be specified to define the set of correct and complete assembly sequences or desired sequences $\mathcal{M}_{\mathcal{J}} \subset \mathcal{M}$.

Definition 2 (Suffix): For $k \geq 0$, *the k-truncated suffix of a temporal sequence* $\sigma = s_0 - s_1 - s_2 - \cdots$ *(written:* $\sigma^{(k)}$*) is the sequence* $s_k - s_{k+1} - s_{k+2} - \cdots$.

$I^{(k)}$ is the interpretation obtained from I, by replacing σ with $\sigma^{(k)}$.

Now, for all interpretations I, we say that I satisfies w (written: $\models^I w$) where the satisfaction relation is defined inductively on formulae as follows:

$\models^I w_1 \vee w_2$ iff $\models^I w_1 \vee \models^I w_2$

$\models^I w_1 \wedge w_2$ iff $\models^I w_1 \wedge \models^I w_2$

$\models^I \neg w$ iff $\neg \models^I w$

$\models^I \Box w$ iff for all $k \geq 0$, $\models^{I^{(k)}} w$.

$\models^I \diamond w$ iff there exists a $k \geq 0$ such that $\models^{I^{(k)}} w$.

$\models^I \bigcirc w$ iff $\models^{I^{(1)}} w$.

$\models^I w_1 \mathcal{U} w_2$ iff there is a $k \geq 0$ such that $\models^{I^{(k)}} w_2$ and for all $i, 0 \leq i < k$, $\models^{I^{(i)}} w_1$.

$\models^I w_1 \mathcal{P} w_2$ iff $\neg \models^I \neg w_1 \mathcal{U} w_2$

Any formula that is not qualified by a temporal operator is assumed to be true only in the initial state s_0.

[3] Global constant: Assigned variable which remains constant over the finite list of states or trajectory.

[4] Our assembly interpretation does not require any global constant. The $\mathcal{G}$-tuple has been introduced only for a more complete definition of Interpretation.

B. Disassembly and R-Equivalence

In assembly sequence planning, two basic approaches are adopted to generate assembly sequences—the assembly and disassembly methods [6], [8]. Establishing transformation properties between assembly and disassembly forms of knowledge motivates the concept of R-equivalence. First, we introduce the notion of *reverse-interpretation.*

Definition (Reverse-Interpretation $\overline{I}$*): Given an interpretation I, its reverse - interpretation* $\overline{I}$ *is a* 3-*tuple* ($\overline{\mathcal{M}}$,$\mathcal{G}$, $\overline{\sigma}$), *where* $\overline{\sigma}$ *is an exact reverse of* σ *such that* $\overline{\mathcal{M}}$ *is similar to* $\mathcal{M}$ *except that all its transitions reverse directions.*

Note that for a given assembly, $\mathcal{M}$ is defined and known. An assembly interpretation (denoted by $I_{\mathcal{A}}$) can therefore be simply written as σ, and a reverse-assembly or disassembly interpretation (denoted by $\overline{I}_{\mathcal{A}}$ or $I_{\mathcal{D}}$) can be written as $\overline{\sigma}$.

1) R-equivalence: Given two symbolic logic functions (formulae) of precedence knowledge, $F = f(p_1, p_2, p_3, \cdots, p_n)$ and $G = g(p_1, p_2, p_3, \cdots, p_n)$, where $p_1, p_2, p_3, \cdots, p_n$ are logic variables, G is said to be R-equivalent to F if and only if, for all I,

$$\models^I F \equiv \models^{\overline{I}} G \tag{1}$$

Definition 4 (Sequence-Invariance): A temporal logic formula H is said to be sequence-invariant if, and only if, for all I,

$$\models^I H \equiv \models^{\overline{I}} H \tag{2}$$

Clearly, a sequence-invariant H is R-equivalent to itself and the following theorem dictates that the set of H is not empty.

Theorem 1 (Invariance Theorem): A temporal logic formula H is sequence-invariant if it is a simple logic combination of terms, each of which is of the form $\Box Q$ *or* $\diamond Q$, *where Q is any classical state-formula (i.e., without temporal operators).* ♣

Proof: see Section IX.

A sufficient condition for the formula F to be R-equivalent to formula G is that there exists a sequence-invariant H such that for all I,

$$\models^I (F \equiv H) \tag{3}$$

and

$$\models^{\overline{I}} (G \equiv H) \tag{4}$$

Hence, if we could "discover" any sequence-invariant H such that (3) and (4) hold for $I = I_{\mathcal{A}}$, we have in fact found one transformation property between assembly and disassembly.

Since assembly tasks are not necessarily reversible, the R-equivalence of the assembly and disassembly methods will hold only if each task used in disassembly is the inverse of a feasible assembly task, regardless of whether this inverse task is itself feasible or not. The term "disassembly task" therefore refers to the logical inverse of a feasible assembly task.

III. Temporal Formulation of Task-Sequence Behaviour

An assembly process-task (or simply, assembly task) refers to an exclusive abstract entity whose execution results in the involved assembly parts being installed in their final relative

positions. Before stating the state-theorems and process axioms for assembly and disassembly, some important terminologies are first defined.

Definition 5 (Assembly process task-symbol): It is a boolean or logic variable p that indicates whether an assembly task it represents has been done or not.

A more general term to assert the DONE status of a group of tasks is given by the following definition.

Definition 6 (Assembly process task-formula): It is any formula P that contains at least one assembly process task-symbol and satisfies the following conditions:

1) *It does not have any "$\neg$" connective.*
2) It does not have any temporal operators.

Therefore, an assembly process task-formula is a simple logic expression, in positive form, of $\wedge$s and $\vee$s only, relating the process task-symbols.

A. Process State-Theorems

Let $\{p_1, p_2, \cdots, p_n\}$ be a set of n distinct process task-symbols that constitute the whole assembly. Let no two task-symbols correspond to the same assembly task.

1) Initial state-theorem (NOR Theorem): In the initial state of any assembly sequence, all assembly tasks are not done, that is:

$$\sum_{i=1}^{n} p_i = \text{ False}$$

2) Final state-theorem (AND Theorem): In the final state of any assembly sequence, all assembly tasks are done, that is:

$$\prod_{i=1}^{n} p_i = \text{ True}$$

Note that in the disassembly process, the AND theorem becomes the initial state-theorem and the NOR theorem becomes the final state-theorem.

B. Process-Axioms

Suppose P is any assembly process task-formula. Then:

APX1 : $\vdash \Box\,(P \rightarrow \Box\, P)$ (Task-persistence property)
APX2 : $\vdash \diamond\, P$ (Task-liveness property)

Intuitively, APX1 may be paraphrased as "whenever P becomes true in an assembly process state, it will always remain true in subsequent states of the assembly process." APX2 may be interpreted as "there is a future assembly process state in which P is true."

APX1 and APX2 are necessity properties. APX1 constitutes a necessary condition that for process efficiency, in robotic assembly for example, every assembly task, once done, should not be undone or redone at all throughout the rest of the assembly process. The "DONE" status of every assembly task should persist, hence it is called the task-persistence property. APX2 constitutes the nature of the assembly process in that every assembly task should eventually be executed for a complete assembly. This is "something good" that must occur, and hence it is called the task-liveness property. Collectively, these properties imply that every assembly task must be done once and only once.

Together with the NOR theorem, these two axioms form a general axiom set[5] or domain $\mathcal{A}$ which characterizes the fundamental nature of assembly sequences. A logical reverse of the assembly process is the disassembly process, characterized by the general set $\mathcal{D}$, which is defined by the AND theorem and the following "reverse" axioms:

$$\text{DPX1} :\vdash \Box\,(\neg P \rightarrow \Box \neg P)$$
$$\text{DPX2} :\vdash \diamond\, \neg P$$

Now, by T80 in Section X of [19]:

$$\vdash \Box\,(P \rightarrow \Box\, P) \wedge \diamond\, P \equiv \neg P \mathcal{U} \Box\, P \tag{5}$$

Thus the "condensed" axiom for assembly is:

$$APCX :\vdash \neg P \mathcal{U} \Box\, P$$

Intuitively, APCX may be paraphrased as "P is not true (or done) until an eventual state when it becomes true and remains so in subsequent states of the assembly process."

By replacing P in (5) by $\neg P$, we have the following "condensed" axiom for disassembly:

$$DPCX :\vdash P \mathcal{U} \Box \neg P$$

Intuitively, DPCX may be paraphrased as "P is true until an eventual state when it becomes false (or undone) and remains so in subsequent states of the disassembly process."

With these fundamental axioms, we may formally define an assembly process-state, a legal assembly sequence and its corresponding disassembly sequence as follows:

Definition 7 (Assembly process-state): It is a discrete instance $s_t \in S$ that assigns to every task-symbol a true or false value to indicate whether the assembly task it represents is done or not done, respectively. It is discrete in that it is the instance at which some m tasks, where $m \geq 0$, begin execution and no other task is being executed.

Definition 8 (Legal assembly interpretation $I_{\mathcal{A}}$ or sequence σ): σ is any finite sequence of different accessible assembly process-states that obeys the assembly process-axioms and can be represented in the following general form:

$$s_0 - s_1 - s_2 - \cdots - s_n, \text{ where n is an integer } \geq 0.$$

s_0 is the initial state where all assembly tasks are not done (ie, all task-symbols (p_j) are false) and s_n is the final state where all the assembly tasks are done (ie, all task-symbols (p_j) become true).

Definition 9 (Legal disassembly interpretation $I_{\mathcal{D}}$ or sequence $\overline{\sigma}$): Given a legal assembly sequence σ, the corresponding disassembly sequence $\overline{\sigma}$ obeys the disassembly process-axioms, and with respect to the different accessible *assembly process-states, is given by:*

$$s_n - s_{n-1} - \cdots - s_2 - s_1 - s_0, \text{ where n is an integer } \geq 0.$$

s_n is the initial state where all assembly tasks are done (ie, all task-symbols (p_j) are true) and s_0 is the final state where all

[5] General axiom set: Axioms that apply to every task-symbol.

the assembly tasks are undone (ie, all task-symbols (p_j) become false).

In task-sequence planning for a manufactured part or assembly, any sequence that is not legal need not be considered further because it is either not process efficient or it will not result in a complete assembly.

Using the process-axioms, many assembly sequence properties could be proved via *mechanical theorem proving* using well-established rules and theorems of temporal logic. See Section B for the assembly sequence properties. The first 12 properties, APT1–APT12, correspond to Huang and Lee's [6] results but proved using temporal logic theory as originally demonstrated in Seow and Devanathan [19], [20]. The two transformation properties, APT 1 and APT 2, are derived by applying the concept of R-equivalence and *Invariance Theorem* as shown in Section X.

IV. Representation and Analysis

A. The Temporal Framework

In our application, we are strictly reasoning in the assembly domain $\mathcal{A}$ in which any legal sequence, if represented as an ordered list, is a sequence of all defined task-symbols that appear once and only once. This is a symbolic view of a legal sequence that conforms to the assembly domain $\mathcal{A}$. Therefore, it is sufficient to specify the model-characteristic temporal constraints[6], with the convention that every task that constitutes the assembly is not done initially and would eventually be executed, once and only once, without having to explicitly state so.

Formally, if $f(p_1,p_2, .., p_n)$ is an assembly constraint, where p_i is a task-symbol, this implies:

$$\vdash^{\mathcal{A}} f(p_1,p_2,\cdots,p_n) = \vdash \left(\prod_{i=1}^{n} \neg p_i \wedge [\neg p_i \mathcal{U} \Box p_i]\right) \wedge f(p_1,p_2,\cdots,p_n)$$

In the assembly domain $\mathcal{A}$, unless otherwise stated or implied, any assembly property or constraint P is simply written as $\vdash P$ or P instead of $\vdash^{\mathcal{A}} P$. We also use the convention that:

$$P_{\mathcal{A}} = f(p_1,p_2,..,p_n) \text{ if, and only if,}$$
$$P = \left(\prod_{i=1}^{n} \neg p_i \wedge [\neg p_i \mathcal{U} \Box p_i]\right) \wedge f(p_1,p_2,\cdots,p_n)$$

Where assembly and disassembly precedence knowledge appear in an expression, the former will be prefixed by $\vdash^{\mathcal{A}}$ and the latter, $\vdash^{\mathcal{D}}$, unless it is obvious from the context.

B. Implicit Representation Scheme

Given an assembly which can be completely connected by a set of n distinct assembly tasks $\{p_1,p_2,...,p_n\}$, an assembly process-state formula $state_i$ asserts a particular assembly state, and is a simple logic function of $\wedge$s only, involving all the task-symbols once, and only once. Formally,

$$state_i = p_a \wedge p_b \cdots \wedge p_h \wedge \neg p_p \wedge \neg p_q \cdots \wedge \neg p_z$$

with

$$\{a,b,\cdots,h\} \cap \{p,q,\cdots,z\} = \{\}$$

and

$$\{a,b,\cdots,h\} \cup \{p,q,\cdots,z\} = \{1,2,\cdots,n\}$$

1) Infeasible Assembly States or Subassemblies:

$$\Box \neg \left(\sum_{i=1}^{j} state_i\right),$$

where $state_i$ is an infeasible assembly process-state formula. This specification may be paraphrased as "every assembly process-state formula $state_i$, for $1 \leq i \leq j$, is not true in every state of the assembly process".

2) Feasible Assembly States or Subassemblies:

$$\prod_{i=1}^{j} \diamond\ state_i,$$

where $state_i$ is a feasible assembly process-state formula. This specification may be paraphrased as "each assembly process-state formula $state_i$, for $1 \leq i \leq$ j, must eventually become true in some state of the assembly process."

3) Precedence constraints: In general, precedences are described in any of the following forms:

$$(A) \quad \cdot X$$
$$(B) \quad X \rightarrow Y$$
$$(C) \quad \Box (X \rightarrow Y)$$

where X and Y are any appropriate temporal logic formulae.

Such a representation scheme, with built-in expressiveness as demonstrated, enables many temporal constraints to be described, limited only by the experience of the designer.

C. Analysis: Selection by Logic Integration

Given that $\mathcal{H}$ is a temporal formula of minimal ordering or geometric constraints [2] inherent to a given assembly, $\mathcal{M}_{\mathcal{H}} \subset \mathcal{M}$ describes the largest set of all and only possible assembly sequences.

Suppose a user-specified soft-constraint formula $\mathcal{S}$ corresponds to $\mathcal{M}_{\mathcal{S}}$, then $\mathcal{H} \wedge \mathcal{S}$ would correspond to the desired (or feasible) sequences $\mathcal{M}_{\mathcal{H}\wedge\mathcal{S}} \subseteq \mathcal{M}_{\mathcal{H}}$, a set obtained by the intersection of the two sets $\mathcal{M}_{\mathcal{H}}$ and $\mathcal{M}_{\mathcal{S}}$; that is:

$$\mathcal{M}_{\mathcal{H}\wedge\mathcal{S}} = \mathcal{M}_{\mathcal{H}} \cap \mathcal{M}_{\mathcal{S}}$$

Logic integration refers to the $\wedge$ operation of $\mathcal{H}$ and $\mathcal{S}$ to obtain $\mathcal{J}$ (that is, $\mathcal{J} = \mathcal{H} \wedge \mathcal{S}$). By applying the assembly sequence properties, it is possible to obtain a more compact representation for $\mathcal{J}$.

$\mathcal{M}_{\mathcal{H}\wedge\mathcal{S}}$ can be directly computed by using the derived assembly sequence properties and the automaton model of $\vdash^{\mathcal{A}} p_j \mathcal{P} p_k$, as presented in the next section.

[6] Model-characteristic constraints: specific assembly constraints to be satisfied by any LEGAL sequence of a *particular* assembly for feasibility.

TABLE II
SYMBOL REDEFINITIONS

No.	Symbolic representations in	
	Theoretical framework	Practical Framework
1	$P_1 \mathcal{P} P_2$	precede(L_P_1, L_P_2)
2	$\neg P_2 \mathcal{U} P_1$	not_until(L_P_2, L_P_1)
3	$\Box\neg(State)$	always_not(L_State)
4	$\Diamond(state_i)$	eve(L_state_i)
5	$\wedge$	,
6	$\vee$	;

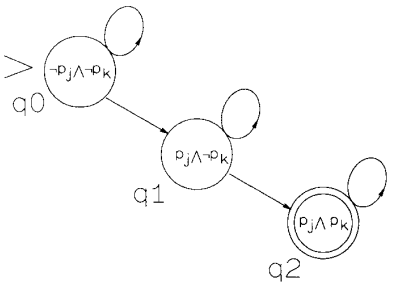

Fig. 1. Automaton for the satisfaction relation of $\vdash^{\mathcal{A}} p_j \mathcal{P} p_k$, where p_j and p_k are assembly task-symbols.

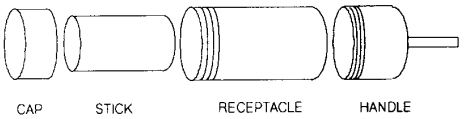

Fig. 2. A simple product in exploded view.

V. SOFTWARE IMPLEMENTATION

Based on the theoretical framework, a simple verification and synthesis tool, called the Mechanical Assembly Sequence Satisfiability Checker (MASS-C), has been built using *Quintus* PROLOG [21] on the *IPC SUN SPARC* workstation. The temporal operators are implemented as logic predicates (see Table II, redefinition), where L_X is a symbolic prefix list notation for the formula X so that for any assembly, the formulated temporal formulae, asserting the acquired assembly constraints, can be directly coded as a logic program of the respective predicates. Auxiliary predicates (*verify()* and *generate()*) are implemented to verify or synthesize all assembly sequences $\mathcal{M}_{\mathcal{H}\wedge\mathcal{S}}$ that conform to this logic program of assembly constraints $\mathcal{H} \wedge \mathcal{S}$.

The current MASS-C is built based on the definition for $\mathcal{P}$ and the properties APT 4, 6, 8, 10, 15 and 16 (see Section X), using recursion [22] as the basic design strategy, with the list structure for representing the explicit assembly sequences, both serial and parallel, as well as the assembly process task-formula in prefix list form, since list is well-suited for PROLOG recursive programs. As an ordered list of tasks, a parallel or concurrent sequence has at least one member which is a list subset containing 2 or more task-symbols representing concurrent tasks. A serial sequence is just an ordered list whose members are single task-symbols.

Using these properties, a useful class of temporal expressions for assembly application can be expressed in terms of the basic element $p_j \mathcal{P} p_k$. Therefore, the finite state-graph structure or automaton for the restricted satisfaction relation of this basic expression, as shown in Fig. 1, suffices for our implementation purpose. It forms the basic underlying invariant structure upon which MASS-C is built.

To interpret the structure, consider any finite sequence σ of discrete assembly process-states whose tasks include p_j and p_k. If σ starts in node q_0, traverses along the transitions, and ends in the accepting node q_2, represented by a double concentric circle, it is said to satisfy this basic assembly constraint $p_j \mathcal{P} p_k$. The reader is referred to Seow [23] for actual implementation details.

The implementation has been carried out using the generate and test (GT) paradigm [24]. In this paradigm, each possible combination of assembly tasks is systematically generated and then tested to see if it satisfies the integrated constraint $\mathcal{H} \wedge \mathcal{S}$ by using recursively the satisfaction relation of the basic constraint mentioned above. In order to find the serial assembly sequences, the number of combinations considered by this method is the size of the cartesian product of the domains of all the task variables. For a serial assembly sequence consisting of n tasks, the domain size of each task variable of the ordered list decreases uniformly as $n, (n-1), \cdots, 1$ along the sequence. Hence the total number of combinations to be considered is $n!$. However, more computationally efficient methods exist for solving the problem. For example, given a constraint $p_1 \mathcal{P} p_2$, instantiation of the task symbol p_2 to the $1st$ task variable will result in an immediate failure of the combination in satisfying the constraint. More generally, those combinations that instantiate the symbol p_2 to the ith task variable will fail when a jth task variable is instantiated the symbol p_1 in an ordered list where $i < j$ for all instantiations of the kth task variables for $i < k < j$. Using a certain algorithm for propagating the constraints to the domains of the different task variables, efficient search techniques for the constraint satisfaction problem result [24]. For a uniform domain size d for each task variable, and the total number of binary constraints (between two task symbols) being e, the worst case complexity of the algorithm is $\mathbf{O}(e \times d^2)$.

VI. EXAMPLE

An exploded view of the assembly taken from [10] is shown in Fig. 2. This example is chosen because it is simple and so does not obscure the illustration of the analysis. It also facilitates comparison with the AND/OR graph representation [10] as discussed under Section VII.

Three distinct assembly tasks to be performed are defined as given below:

1) p_1: screw task, connecting CAP and RECEPTACLE.
2) p_2: screw task, connecting RECEPTACLE and HANDLE.
3) p_3: insert task, connecting RECEPTACLE and STICK.

A. Hard-Constraint Specification

The geometric constraint $\mathcal{H}$ that the stick must be in the receptacle before both ends are attached is expressed as follows:

$$\mathcal{H} = (\neg p_1 \wedge \neg p_2 \wedge \neg p_3) \wedge (\neg p_1 \mathcal{U} \Box p_1 \wedge \neg p_2 \mathcal{U} \Box p_2 \wedge \neg p_3 \mathcal{U} \Box p_3) \wedge \underline{p_3 \mathcal{P}(p_1 \wedge p_2)}$$

where the model-characteristic hard constraint $\mathcal{H}_\mathcal{A}$ is:

$$\begin{aligned}\mathcal{H}_\mathcal{A} &= p_3 \mathcal{P}(p_1 \wedge p_2) \\ &= p_3 \mathcal{P} p_1 \vee p_3 \mathcal{P} p_2 \text{ by APT8}\end{aligned}$$

The following is an enumeration of the serial (No. 1–4) and parallel (No. 5–7) sequences in $\mathcal{M}_\mathcal{H}$. The notations $[p_i, p_j]$ and $< p_i, p_j >$ represent that the tasks p_i and p_j are done sequentially and in parallel, respectively.

Sequence Number	Sequence Ordered List
1	$[p_1, p_3, p_2]$
2	$[p_2, p_3, p_1]$
3	$[p_3, p_1, p_2]$
4	$[p_3, p_2, p_1]$
5	$[p_3, < p_1, p_2 >]$
6	$[< p_1, p_3 >, p_2]$
7	$[< p_2, p_3 >, p_1]$

B. Soft-Constraint Specification

1. $\mathcal{S}_1 = \Box \neg(\neg p_1 \wedge \neg p_2 \wedge p_3)$
2. $\mathcal{S}_2 = \diamond\ (\neg p_1 \wedge \neg p_3 \wedge p_2)$
3. $\mathcal{S}_3 = p_1 \mathcal{P} p_3$

   $\mathcal{S}_1$ specifies that the assembly process-state $(\neg p_1 \wedge \neg p_2 \wedge p_3)$ must not appear in the sequence, $\mathcal{S}_2$ specifies that $(\neg p_1 \wedge \neg p_3 \wedge p_2)$ must be one of the assembly process-states in the sequence, and $\mathcal{S}_3$ specifies that p_1 must precede p_3 in the assembly sequence.
4. $\mathcal{S}_1$ is to be executed serially only as in the following specification:

$$\mathcal{S} = (p_1 \mathcal{P} p_2 \vee p_2 \mathcal{P} p_1) \wedge (p_2 \mathcal{P} p_3 \vee p_3 \mathcal{P} p_2) \wedge (p_1 \mathcal{P} p_3 \vee p_3 \mathcal{P} p_1) \wedge \mathcal{S}_1$$

which in shorthand notation can be written as:

$$\mathcal{S}_G = \Box \neg(\neg p_1 \wedge \neg p_2 \wedge p_3), \text{ where } G = p_a \mathcal{P} p_b \vee p_b \mathcal{P} p_a, \text{ and } p_a, p_b \text{ are distinct .}$$

C. Logic Analysis: Integration and Simplification

Assembly sequence properties derived (see Section X) are applied to simplify the integrated specifications $\mathcal{J}_\mathcal{A}$ which could result in a more compact representation.

For **case 1:**

$$\begin{aligned}\mathcal{S}_1 &= \Box \neg(\neg p_1 \wedge \neg p_2 \wedge p_3) \\ &= \Box\ (\neg p_3 \vee (p_1 \vee p_2)) && \text{by DeMorgan's Law} \\ &= \neg p_3 \mathcal{U} (p_1 \vee p_2) && \text{by APT15} \\ &= (\neg p_3 \mathcal{U} p_1) \vee (\neg p_3 \mathcal{U} p_2) && \text{by APT 5}\end{aligned}$$

Therefore,

$$\begin{aligned}\mathcal{J}_\mathcal{A} &= \mathcal{H}_\mathcal{A} \wedge \mathcal{S}_1 \\ &= (p_3 \mathcal{P} p_1 \vee p_3 \mathcal{P} p_2) \wedge (\neg p_3 \mathcal{U} p_1 \vee \neg p_3 \mathcal{U} p_2) \\ &= \underline{(p_3 \mathcal{P} p_1) \wedge (\neg p_3 \mathcal{U} p_1)} \vee \underline{(p_3 \mathcal{P} p_2) \wedge (\neg p_3 \mathcal{U} p_2)} \vee \\ &\quad (p_3 \mathcal{P} p_2) \wedge (\neg p_3 \mathcal{U} p_1) \vee (p_3 \mathcal{P} p_1) \wedge (\neg p_3 \mathcal{U} p_2) \quad \text{by PR}^7 \\ &= (p_3 \mathcal{P} p_2) \wedge (\neg p_3 \mathcal{U} p_1) \vee (p_3 \mathcal{P} p_1) \wedge (\neg p_3 \mathcal{U} p_2) \text{ by APT22.}\end{aligned}$$

By considering the two sum of product (SOP) terms, the set of selected sequences, $\mathcal{M}_{\mathcal{H} \wedge \mathcal{S}}$, contains those in $\mathcal{M}_\mathcal{H}$ with sequence numbers 1, 2, 6, and 7.

By a similar reasoning , for **case 2**, $\mathcal{S}_2$ is simplified to:

$$\mathcal{S}_2 = (p_2 \mathcal{P} p_1) \wedge (p_2 \mathcal{P} p_3)$$

and

$$\begin{aligned}\mathcal{J}_\mathcal{A} &= \mathcal{H}_\mathcal{A} \wedge \mathcal{S}_2 \\ &= (p_2 \mathcal{P} p_3) \wedge (p_3 \mathcal{P} p_1)\end{aligned}$$

The only sequence selected is sequence number 2.

Similarly, **case 3** results in the selected sequence 1 only.

For **case 4**, in short-hand notation,

$$\begin{aligned}\mathcal{J}_{\mathcal{A} \wedge G} &= \mathcal{H}_\mathcal{A} \wedge \mathcal{S}_G \\ &= (p_3 \mathcal{P} p_1 \vee p_3 \mathcal{P} p_2) \wedge \Box \neg(\neg p_1 \wedge \neg p_2 \wedge p_3) \\ &= (p_3 \mathcal{P} p_2) \wedge (\neg p_3 \mathcal{U} p_1) \vee (p_3 \mathcal{P} p_1) \wedge (\neg p_3 \mathcal{U} p_2) \text{ by Case 1}\end{aligned}$$

This means that:

$$\mathcal{J}_\mathcal{A} = (\underline{(p_3 \mathcal{P} p_2) \wedge (\neg p_3 \mathcal{U} p_1)} \vee \underline{(p_3 \mathcal{P} p_1) \wedge (\neg p_3 \mathcal{U} p_2)}) \wedge (p_1 \mathcal{P} p_2 \vee p_2 \mathcal{P} p_1) \wedge (p_2 \mathcal{P} p_3 \vee p_3 \mathcal{P} p_2) \wedge (p_1 \mathcal{P} p_3 \vee p_3 \mathcal{P} p_1)$$

which can be simplified to:

$$\mathcal{J}_\mathcal{A} = (p_1 \mathcal{P} p_3 \wedge p_3 \mathcal{P} p_2) \vee (p_2 \mathcal{P} p_3 \wedge p_3 \mathcal{P} p_1)$$

Hence, the selected sequences are sequence numbers 1 and 2.

The complete temporal model $\mathcal{J}$ for this case is:

$$\begin{aligned}\mathcal{J} = &(\neg p_1 \wedge \neg p_2 \wedge \neg p_3) \\ &\wedge (\neg p_1 \mathcal{U} \Box p_1 \wedge \neg p_2 \mathcal{U} \Box p_2 \wedge \neg p_3 \mathcal{U} \Box p_3) \\ &\wedge (\underline{p_1 \mathcal{P} p_3 \wedge p_3 \mathcal{P} p_2} \vee \underline{p_2 \mathcal{P} p_3 \wedge p_3 \mathcal{P} p_1})\end{aligned}$$

D. Automatic Generation and Verification

Using MASS-C, explicit assembly sequences can be automatically generated or verified based on the given constraints.

The logic program segment for the hard constraint $\mathcal{H}_\mathcal{A}$ is given by:

```
hard_constraint:-
precede (p3, [and, p1, p2]).
```

and that for the soft-constraints $\mathcal{S}$, say $\mathcal{S} = \mathcal{S}_1 \wedge \mathcal{S}_2$, is given by:

```
soft_constraint:-
always_not ([and, [not_and, p1, p2], p3]),
                    /* Infeasible assembly state */
eve([and, [not_and, p1, p3], p2]).
                    /* Necessary assembly state */
```

[7] "PR" stands for "Propositional Reasoning."

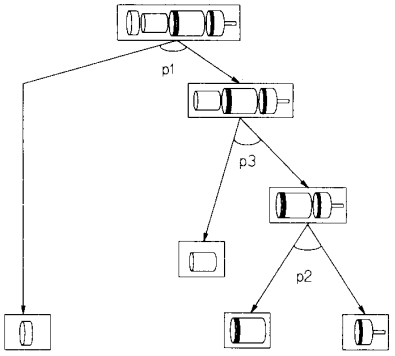

Fig. 3. A solution tree selected by $\mathcal{S}_2$.

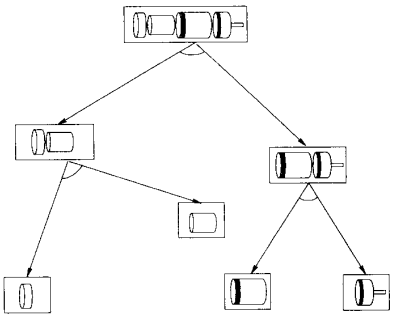

Fig. 4. A solution tree of the AND/OR graph (for Fig. 2) corresponding to more than one assembly sequence.

After successfully compiling the program in the Quintus prolog interpretive mode, we can synthesize or verify the sequences by simply invoking the respective predicates.

To synthesize, type *generate*(Seq). and press ≪Enter≫.

The PROLOG interpreter returns the serial sequence:

Seq = [p2,p3,p1]

no

In our example, only one serial sequence satisfies the constraints.

To verify if the concurrent sequence [< p2, p3 > p1,] satisfies the specified constraints:

Type *verify* ([< p2, p3 > p1,]). and press ≪Enter≫.

The PROLOG interpreter returns the answer:

no

This means that the input sequence does not satisfy the constraints.

VII. Comparison and Discussion

In this section, we attempt to reason that the proposed framework can be considered a generalization of the existing representation schemes whose basic elements are as given in Table I. The example of Section VI is used to illustrate the comparisons made.

In **case 2** under Section VI-C for example, the integration of $\mathcal{S}_2$ with $\mathcal{H}_\mathcal{A}$ using the proposed temporal logic framework yields the enumerated sequence $[p_2, p_3, p_1]$ (sequence number 2). This sequence can be represented as a solution tree of the AND/OR graph (Fig. 4 in Homem de Mello and Sanderson [10]) for the same assembly, as shown in Fig. 3. Qualitatively however, there is no equivalent mechanism in the AND/OR graph representation to "pick" the solution tree satisfying the additional constraint $\mathcal{S}_2$. The advantage of such a mechanism is evident for a more complex product than that given in Fig. 2.

In the AND/OR graph formulation, it is observed that in assemblies where time-independence and concurrency among some tasks are possible (for example, the solution tree in Fig. 4), the AND/OR graph cannot represent the precedence among these tasks due to additional constraints that prohibit concurrency, without additional constructs which may obscure the elegance of the AND/OR representation [2].

Apparently, some sequences from the AND/OR graph (for example, the solution tree in Fig. 4) are missing in our enumeration. This is only because we have identified three tasks, viz, p_1, p_2, and p_3, for the assembly that correspond to the respective connections of the *graph of connections* model for the product (see Fig. 5). The connections, c_1 and c_2, are not treated as tasks but contacts established by the assembly process.

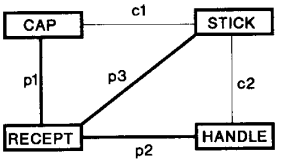

Fig. 5. Graph of connections for the simple product of Fig. 2.

Also under case 2 in Section VI-C, we see that $\mathcal{S}_2$ directly specifies that the assembly process-state described by the state-formula $(\neg p_1 \wedge \neg p_3 \wedge p_2)$ is an eventual state of the sequence. If any of the existing schemes such as Huang and Lee's [6] is used, such a constraint can only be indirectly specified as MP(p_2, $p_1 \vee p_3$). This is certainly not a natural and an easy way of specifying a desired assembly state. Such a limitation is magnified in a larger state problem than the simple example considered here, when expressing one or more desirable state-formulae in terms of the MP(). predicate becomes obviously cumbersome. In the proposed framework, property APT16 allows the equivalent precedence constraint to be directly derived from the desired assembly process-state. This means that the framework actually provides a flexible means to specify and analyze the precedence relationships among the assembly tasks involved. A similar reasoning using property APT15 argues for the Always $\Box$ operator as used in $\mathcal{S}_1$.

Although the serial implicit representations by Fox [2], De Fazio and Whitney [5] and Ayoub and Doty [7] may actually be modified or extended to logically encompass concurrent sequences, their only "Precede" operator does not offer flexibility of expression. For instance, a simple constraint

that can be specified only as $\neg(p_1 < p_2)$, using any of these existing schemes, can actually be re-specified either as $\neg p_1 \mathcal{U} p_2$ or its equivalent $\neg(p_1 \mathcal{P} p_2)$ in the proposed framework. In any case, these existing schemes lack formal treatment in that there is no formal inference mechanism to logically prove the algebraic properties involving their only "Precede" operator.

In the above discussion, it is clearly illustrated that the proposed framework generalizes the existing representation schemes. It provides a richer vocabulary of temporal expressions. In fact, it is even possible to specify a more complex formula such as $p_1 \mathcal{P} p_2 \rightarrow \diamond (State)$, which can be intuitively interpreted as "if p_1 is executed before p_2 in the sequence, then the sequence must contain the process-state $State$ as well." With a strong underlying logic mechanism of general rules and theorems not found in all previous work, assembly sequences can be mechanically analyzed through the use of straightforward syntactic manipulation of precedence and state constraint formulae.

Finally, our framework can also support backward assembly planning. Backward assembly planning is a method of obtaining feasible assembly sequences from feasible disassembly sequences generated [25]. Planning assembly backwards may need to identify the additional constraint for assembly (which can be conveniently classified as soft-constraint in this paper) which is a non-issue in disassembly. Based on a given set of disassembly constraints, we obtain a logic equivalent of the assembly constraints by APT1 and APT2. By identifying and imposing the additional constraints for assembly upon these equivalent assembly constraints, we can obtain the feasible assembly sequences.

VIII. Conclusion

We have proposed temporal logic as an implicit representation tool and analytical language for assembly sequence planning. The strengths of the framework over existing representations lie in its strong manipulability, temporal expressiveness and precise formalism. As a basis, task-persistence and task-liveness properties that describe the general assembly sequence behavior have been formulated. The R-equivalence concept is introduced, relating assembly and disassembly forms of knowledge. Properties that facilitate reasoning and manipulation of a class of precedence knowledge asserted by $\mathcal{P}$, $\mathcal{U}$, $\Box$ and $\diamond$ operators have been proved. Together with the support for disjunctive ordering, $\mathcal{P}$ and $\mathcal{U}$ operators form the theoretical basis for modelling or encoding all possible assembly sequences for a mechanical assembly. Enriched with other operators, temporal logic provides a flexible framework for representation and analysis of temporal constraints of assembly tasks arising from criteria such as the geometric, mechanical and stability properties of a given mechanical assembly. Comparison reveals that the existing representation schemes can be generalized, in terms of the precedence elements, as a language subset of the proposed framework. The framework has been implemented on a workstation for automatic generation of serial assembly sequences. The complexity issues of the computation involved have been briefly discussed. A simple example illustrates the use of the proposed framework.

IX. A Proof of Invariance Theorem

Let Q be any state-formula, I be any arbitrary general interpretation (corresponding to a sequence σ), and $\overline{I}$ be the reverse interpretation (corresponding to an exact reverse-sequence $\overline{\sigma}$). Then:

$\models^I \Box Q$
iff for all $k \geq 0, \models^{I^{(k)}} Q$
iff for every state in σ, Q, is true
iff for every state in $\overline{\sigma}, Q$ is true
iff for all $j \geq 0, \models^{\overline{I}^{(j)}} Q$
iff $\models^{\overline{I}} \Box Q$
Also :
$\models^I \diamond Q$
iff there exists a k ≥ 0 such that $\models^{I^{(k)}} Q$
iff there exists a state in σ in which Q is true
iff there exists a state in $\overline{\sigma}$ in which Q is true
iff there exists a j ≥ 0 such that $\models^{\overline{I}^{(j)}} Q$
iff $\models^{\overline{I}} \diamond Q$

Hence, by induction, if a temporal logic formula H is any logic combination of terms, each of which is of the form $\Box Q$ or $\diamond Q$, then for all I,

$$\models^I H \equiv \models^{\overline{I}} H$$

X. Assembly Sequence Properties

Assume that P_1, P_2, P_3 are any assembly process task-formulae. All stated theorems T_i are restricted to the assembly domain $\mathcal{A}$ and denoted simply as $\vdash T_i$. Where necessary, $\vdash^{\mathcal{A}} T_i$ and $\vdash^{\mathcal{D}} T_i$ would be used to denote that T_i is a theorem, restricted to the assembly domain $\mathcal{A}$ and disassembly domain $\mathcal{D}$, respectively.

APT 1 $\quad \vdash^{\mathcal{A}} \neg P_2 \mathcal{U} P_1 \leftrightarrow \vdash^{\mathcal{D}} P_1 \mathcal{U} \neg P_2$
Proof:
By D 40 [19] $\vdash^{\mathcal{A}} \neg P_2 \mathcal{U} P_1 \leftrightarrow \Box (P_1 \vee \neg P_2)$
Similarly: $\quad \vdash^{\mathcal{D}} P_1 \mathcal{U} \neg P_2 \leftrightarrow \Box (P_1 \vee \neg P_2)$
Note that $\Box (P_1 \vee \neg P_2)$ is a sequence-invariant formula.
Hence: $\quad \vdash^{\mathcal{A}} \neg P_2 \mathcal{U} P_1 \leftrightarrow \vdash^{\mathcal{D}} P_1 \mathcal{U} \neg P_2$

APT 2 $\quad \vdash^{\mathcal{A}} P_2 \mathcal{P} P_1 \leftrightarrow \vdash^{\mathcal{D}} \neg P_1 \mathcal{P} \neg P_2$
Proof:
By definition: $\quad P_2 \mathcal{P} P_1 \leftrightarrow \neg(\neg P_2 \mathcal{U} P_1)$
By T5 [16]: $\quad \neg \Box (P_1 \vee \neg P_2) \leftrightarrow \diamond \neg (P_1 \vee \neg P_2)$
By D40 [19]: $\quad \vdash^{\mathcal{A}} \neg P_2 \mathcal{U} P_1 \leftrightarrow \Box (P_1 \vee \neg P_2)$
Negate both sides: $\quad \vdash^{\mathcal{A}} P_2 \mathcal{P} P_1 \leftrightarrow \diamond \neg (P_1 \vee \neg P_2)$
Similarly: $\quad \vdash^{\mathcal{D}} \neg P_1 \mathcal{P} \neg P_2 \leftrightarrow \diamond \neg (P_1 \vee \neg P_2)$
$\diamond \neg (P_1 \vee \neg P_2)$ is a sequence-invariant formula.
Hence: $\quad \vdash^{\mathcal{A}} P_2 \mathcal{P} P_1 \leftrightarrow \vdash^{\mathcal{D}} \neg P_1 \mathcal{P} \neg P_2$

APT 3 $\quad \vdash (\neg P_1 \wedge \neg P_2) \mathcal{U} P_3 \leftrightarrow \neg P_1 \mathcal{U} P_3 \wedge \neg P_2 \mathcal{U} P_3$
APT 4 $\quad \vdash (P_1 \vee P_2) \mathcal{P} P_3 \leftrightarrow P_1 \mathcal{P} P_3 \vee P_2 \mathcal{P} P_3$
APT 5 $\quad \vdash \neg P_3 \mathcal{U} (P_1 \vee P_2) \leftrightarrow \neg P_3 \mathcal{U} P_1 \vee \neg P_3 \mathcal{U} P_2$
APT 6 $\quad \vdash P_3 \mathcal{P} (P_1 \vee P_2) \leftrightarrow P_3 \mathcal{P} P_1 \wedge P_3 \mathcal{P} P_2$
APT 7 $\quad \vdash \neg P_3 \mathcal{U} (P_1 \wedge P_2) \leftrightarrow \neg P_3 \mathcal{U} P_1 \wedge \neg P_3 \mathcal{U} P_2$
APT 8 $\quad \vdash P_3 \mathcal{P} (P_1 \wedge P_2) \leftrightarrow P_3 \mathcal{P} P_1 \vee P_3 \mathcal{P} P_2$
APT 9 $\quad \vdash (\neg P_1 \vee \neg P_2) \mathcal{U} P_3 \leftrightarrow \neg P_1 \mathcal{U} P_3 \vee \neg P_2 \mathcal{U} P_3$
APT10 $\quad \vdash (P_1 \wedge P_2) \mathcal{P} P_3 \leftrightarrow P_1 \mathcal{P} P_3 \wedge P_2 \mathcal{P} P_3$

APT11 $\vdash \neg P_3 \mathcal{U} P_2 \wedge \neg P_2 \mathcal{U} P_1 \rightarrow \neg P_3 \mathcal{U} P_1$
APT12 $\vdash P_1 \mathcal{P} P_2 \wedge P_2 \mathcal{P} P_3 \rightarrow P_1 \mathcal{P} P_3$
APT13 $\vdash \neg P_1 \mathcal{U} P_1$
APT14 $\vdash \neg(P_1 \mathcal{P} P_1)$
APT15 $\vdash \neg P_2 \mathcal{U} P_1 \leftrightarrow \Box (P_1 \vee \neg P_2)$
APT16 $\vdash P_2 \mathcal{P} P_1 \leftrightarrow \diamond (\neg P_1 \wedge P_2)$
APT17 $\vdash P_2 \mathcal{P} P_1 \rightarrow \neg P_1 \mathcal{U} P_2$
APT18 $\vdash \neg(P_1 \mathcal{P} P_2 \wedge P_2 \mathcal{P} P_1)$
APT19 $\vdash \neg(P_1 \mathcal{P} P_2 \wedge P_2 \mathcal{P} P_3 \wedge P_3 \mathcal{P} P_1)$
APT20 $\vdash \neg P_1 \mathcal{U} P_2 \wedge P_2 \mathcal{P} P_1 \leftrightarrow P_2 \mathcal{P} P_1$
APT21 $\vdash P_1 \mathcal{P} P_3 \wedge P_3 \mathcal{P} P_2 \wedge P_1 \mathcal{P} P_2 \leftrightarrow P_1 \mathcal{P} P_3 \wedge P_3 \mathcal{P} P_2$
APT22 $\vdash \neg(P_1 \mathcal{P} P_2 \wedge \neg P_1 \mathcal{U} P_2)$

For the proofs of APT3–APT12, see Seow and Devanathan [20]. For the proofs of APT13–APT22, see Seow and Devanathan [19]. All proofs may be found in Seow [23].

REFERENCES

[1] A. C. Sanderson and L. S. Homem de Mello, "Automatic generation of mechanical assembly sequences," *Geometric Modeling for Product Engineering*, M. Wozney, J. Turner, and K. Preiss, Eds. New York: Elsevier, 1990, pp. 461–482.

[2] B. R. Fox, "A representation for serial robotic tasks," Ph.D. thesis, Computer Science, University of Missouri-Rolla, 1987.

[3] Alain Bourjault, "Contribution àà une approache méthodologique de l'assemblage automatisé: Elaboration automatique des séquences opératoires." Thése de Doctorat, Université de Besançon Franche-Comté, 1984.

[4] L. S. Homem de Mello and A. C. Sanderson, "Representations of mechanical assembly sequences," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 2, pp. 211–227, April 1991.

[5] T. L. De Fazio and D. E. Whitney, "Simplified generation of all mechanical assembly sequences," *IEEE Journal of Robotics and Automation*, vol. 3, no. 6, pp. 640–658, 1987. Corrections. *IEEE Journal of Robotics and Automation*, vol. 4, no. 6, pp. 705–708, 1988.

[6] Y. F. Huang and C. S. G. Lee, "Precedence knowledge in future mating operation assembly planning," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 216–221, 1989.

[7] R. G. Ayoub and K. L. Doty, "A representation for discrete assembly sequences in task planning," *Proceedings of the IEEE 13th Annual International Computer Software and Applications Conference*, pp. 746–753, 1989.

[8] A. C. Sanderson, L. S. Homem de Mello, and H. Zhang, "Assembly sequence planning," *Artificial Intelligence Magazine*, vol. 11, no. 1, pp. 62–81, 1990.

[9] D. F. Baldwin, T. E. Abell, M. C. M. Lui, T. L. De Fazio, and D. E. Whitney, "An integrated computer-aid for generating and evaluating assembly sequences for mechanical products," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 1, pp. 78–94, 1991.

[10] L. S. Homem de Mello and A. C. Sanderson, "And/or graph representation of assembly plans," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 2, pp. 188–199, 1990.

[11] L. S. Homem de Mello and A. C. Sanderson, "Evaluation and selection of assembly plans," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 1588–1593, 1990.

[12] T. E. Abell, "An interactive software tool for editing and evaluating mechanical assembly sequences based on fixturing and orientation requirements," Master's thesis in Mechanical Engineering, M.I.T., 1989.

[13] Jean-Michel Henrioud and Alain Bourjault, "LEGA: a computer-aided generator of assembly plans," *Computer-Aided Mechanical Assembly Planning*, L. S. Homem de Mello and S. Lee, Eds. Kluwer Academic Publishers, 1991, pp. 341–381.

[14] A. Delchambre, "A pragmatic approach to computer-aided assembly planning," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 1600–1605, 1990.

[15] Z. Manna and A. Pnueli, "A temporal proof system," *Foundations of Computer Science IV, Distributed Systems: Part 2, Semantics and Logic*, J. de Bakker and J. V. Leeuwen, Eds. Mathematical Centre Tracts 159, Amsterdam, 1983, pp. 163–235.

[16] J. S. Ostroff, "Appendix A: Formal overview of temporal logic" and "Section X: Temporal logic theorems and rules," *Temporal logic for Real Time Systems*. New York: Research Studies Press, Ltd., John Wiley & Sons, Inc., Advanced Software Development Series, 1989, pp. 155–171 and 172–183.

[17] L. S. Homem de Mello and A. C. Sanderson, "A correct and complete algorithm for the generation of mechanical assembly sequences," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 2, pp. 228–240, 1991.

[18] E. M. Clarke, M. C. Brown, E. A. Emerson, and A. P. Sistla, "Using temporal logic for automatic verification of finite state systems," *Logics and Models of Concurrent Systems. NATO ASI series, Vol. F13*, K. Apt, Ed. Berlin, Heidelberg: Springer Verlag, 1985, pp. 3–26.

[19] Kiam Tian Seow and R. Devanathan, "A temporal logic framework for assembly sequence planning," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, North Carolina, U.S.A., 1992.

[20] Kiam Tian Seow and R. Devanathan, "Temporal logic formulation of assembly sequence properties," *Proceedings of the IEEE International Conference on Robotics and Automation*, Nice, France, 1992, pp. 1208–1213.

[21] Quintus Corporation, an Intergraph Corporation, *Quintus Prolog Release 3.1.1 for the SUN 3/4, II: Language and Library Manual*, February 1991.

[22] R. Bharath, *PROLOG: Sophisticated applications in Artificial Intelligence*, first ed. Windcrest Books, Inc., ch. 1, "Knowledge representation in prolog" and ch. 2, "Lists and recursion: Key examples," 1989.

[23] Kiam Tian Seow, "Robotic-task sequence representation and analysis," Master of Engineering (M.Eng.) Thesis, School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, June 1992.

[24] Vipin Kumar, "Algorithms for constraint-satisfaction problems: A survey," *Artificial Intelligence Magazine*, vol. 13, no. 1, pp. 32–44, 1992.

[25] S. Lee, "Backward assembly planning with DFA analysis," *Computer-Aided Mechanical Assembly Planning*, L. S. Homem de Mello and S. Lee, Eds. Kluwer Academic Publishers, 1991, pp. 341–381.

Rajagopalan Devanathan, (M'80–SM'91), received the M.Sc. and Ph.D. degrees in Electrical Engineering from Queen's University, Kingston, Ontario, Canada, in 1968 and 1972 respectively, Prior to that, he obtained his M.E. (Power) and B.E. (Electrical Technology) degrees from the Indian Institute of Science, Bangalore, India, in 1964 and 1962, respectively, and B.Sc. degree from the University of Madras, India, in 1959.

He has served in the government of India in the areas of planning and technology development in the fields of mass and space communications and the oil & gas industry. He has also developed and taught courses in control engineering relating to the Petroleum industry. Since 1983 he has been with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His current interests are in intelligent tuning, robotic assembly, control theory and applications.

Dr. Devanathan has served as the General Chairman of the IEEE Singapore International Conference on Intelligent Control and Instrumentation, Singapore, 1992. He has also served as the Editor of the IFAC sysmposium on Intelligent Turning and Adaptive Control, Singapore, 1991.

Kiam Tian Seow graduated with a B.Eng. degree in Computer Engineering from the National University of Singapore (NUS), Republic of Singapore, in April 1990. After a short working stint with the Institute of Systems Science of NUS (ISS-NUS) as a software engineer, he joined the Nanyang Technological University (NTU), Republic of Singapore, in November 1990, where he received his M.Eng. degree in June 1992 under a research assistantship. He is currently a full-time doctoral candidate with the same University under a similar award, working on the control of discrete event systems. His current research interests include qualitative control of discrete event systems and assembly sequence planning.