

References

- [1] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [2] G. Attardi and M. Simi. A description-oriented logic for building knowledge bases. *Proceedings of IEEE*, 74(10):1335–1344, 1986.
- [3] M. Balaban and N.V. Murray. The logic of time structures: Temporal and nonmonotonic features. In *IJCAI-89*, pages 1285–1290, 1989.
- [4] M. Balaban and N.V. Murray. Interleaving time and structure. Technical Report TR 93-14, Department of Mathematics and Computer Science, Ben-Gurion University, Israel, and Department of Computer Science, SUNYA, 1993.
- [5] M.C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: A graph-theoretic approach. Technical Report TR 91-54, DIMACS Center for Discrete Mathematics and Theoretical Computer Science, NJ, 1991.
- [6] M. Kifer, W. Kim, and Y. Sagiv. Querying object oriented databases. In *ACM SIGMOD*, pages 393–402, 1992.
- [7] K.V. Luck, B. Nebel, C. Peltason, and A. Schmiesel. The anatomy of the back system. Technical Report KIT Report 41, Department of Computer Science, Technische Universität Berlin, Berlin, FRG, 1987.
- [8] R. MacGregor. Inside the loom description classifier. *SIGART Bulletin*, 2(3):88–92, 1991.
- [9] P.F. Patel-Schneider and B. Swartout. Description logic specification – from the krss effort. Technical report, AT&T Bell Labs, June 1993.
- [10] L.A. Resnick, A. Borgida, R.J. Brachman, D.L. McGuinness, and P.F. Patel-Schneider. Classic description and reference manual for common lisp implementation. Technical Report Version 1.02, AT&T Bell Labs, 1990.
- [11] W.A. Woods and J.G. Schmolze. The kl-one family. *Computers and Mathematics with Applications*, Special Issue on Semantic Networks in Artificial Intelligence, 1992.

The algorithm terminates, since the number of paths in the graph of p is finite, as is the length of the paths. The resulting graph of e is a model for p by construction. Also, for any plans p_1, p_2 which are descendents of p (p_1 may be p itself), let SC be an arbitrary proper positive constraint (with source p_1 and sink p_2). Let events e_1, e_2 such that $\Lambda(e_1) = p_1$, $\Lambda(e_2) = p_2$, and $start(e_2) - start(e_1) = delay(SC)$. Then there is no event $e_3 \neq p_2$ such that $\Lambda(e_3) = p_2$ that has the same start time as e_2 , by construction. Let U_{e_1} be the set of paths u with source e_1 , and $\Lambda(sink(u)) = p_2$. Then if $\Lambda(u) \in SC$, then $delay(u) = delay(SC) = start(e_2) - start(e_1)$. Since e_2 is the only event mapped to p_2 with the requisite delay from e_1 , then $sink(u) = e_2$. Thus, e satisfies arbitrary proper positive constraint SC , and thus satisfies any proper positive constraint system.

(\rightarrow) To show that consistency implies that constraints are proper, it is sufficient to show that if e is a model for p that satisfies the sharing constraint system, then any two distinct paths w_1 and w_2 in the same positive constraint SC have the same delay. Let p_s be the source of SC , and e_s be any event such that $\Lambda(e_s) = p_s$ (it must exist, or e is not a model of p). Then there must exist two distinct paths u_1, u_2 such that $\Lambda(u_1) = w_1$ and $\Lambda(u_2) = w_2$, and such that $source(u_1) = source(u_2) = e_s$. In order for e to satisfy the constraint SC , there must exist an event $e_e = sink(u_1) = sink(u_2)$ such that $\Lambda(e_e) = sink(SC) = sink(w_1) = sink(w_2)$. The delays of w_1 and w_2 are equal to that of u_1 and u_2 respectively. But the delay of u_1 is equal to that of u_2 , since they have the source and sink events, and thus $delay(w_1) = delay(w_2)$.

No assumption was made in the antecedent, other than that p has a model e , and that w_1, w_2 belong to the same positive sharing constraint. Therefore, the fact that the delays of w_1, w_2 are equal implies that in order for a constraint to be have a satisfying model, it must be proper.

□

Proofs of Theorems

Proof: (\leftarrow) To show that every proper positive constraint system is consistent, we show that it is satisfiable, i.e. that there is a model e for p that satisfies all the constraints. The proof is constructive, i.e. an algorithm that, given a plan p , constructs an event e such that $\Lambda(e) = p$, which satisfies any possible proper positive sharing constraint. We do that by forcing events to be equal if at all possible. The algorithm keeps a queue of (p', t, e') pairs (initially empty), to process components of p top down. For each plan p' , a component of p , a list of $(t, \gamma(p', t))$ pairs are kept. The algorithm constructs the function $e' = \gamma(p', t)$, such that $\Lambda(e') = p'$, during the run⁵. All lists are initially empty.

Algorithm .1 • *Input: a non-elementary plan p .*

• *Output: a graph corresponding to a structured event e .*

1. *Create a new node e , add $(0, e)$ to the list for p , and Expand $(p, 0, e)$.*
2. *While the queue is not empty, get an item (p', t, e') from the (head of the) queue and expand it.*

Expanding an item (p', t) is done as follows:

• *For each element (p', i, p^i, t_i) of p' , do the following (the p^i may or may not be distinct):*

1. *If the list for p^i contains a pair $(t + t_i, e^i)$ then create an arc (e', i, e^i) as an element of e' .*

2. *Otherwise, do the following:*

(a) *Create a new node e^i*

(b) *Create the arc (e', i, e^i) as an element of e'*

(c) *Set $\text{start}(e^i)$ to be $t + t_i$.*

(d) *Add the pair $(t + t_i, e^i)$ to the list for p^i*

(e) *If p^i is an elementary plan, then let e^i be an elementary event, with duration and action equal to those of p^i .*

(f) *Otherwise, insert $(p^i, t + t_i, e^i)$ into tail of the queue.*

⁵This is not exactly the function Γ , since its value is a single event, rather than a set, but is otherwise the same. We have only a single event here by construction.

7 Appendix

in $\Psi(\langle name\ term \rangle)$. If ce_i is a positive C-expression, it is sufficient to check whether all paths in $\tilde{\Psi}(ce_i)$ have a common delay.

Given a semantic structure in which the above plan definition holds, the following algorithm (sketched) checks for positive ce_i -s ($0 \leq i \leq n$), whether ce_i is satisfied.

Algorithm 4.2 • **Input:** (\mathcal{P}, Ψ) , a semantic structure, ce , a positive C-expression, $p = \Psi(\mathbf{head}(ce))$.

• **Output:** YES - if $(\mathcal{P}, \Psi) \models ce$; NO - otherwise.

• **Method:**

1. Systematically generate all plan paths in $\Psi(\mathbf{head}(ce))$, with head $\Psi(\mathbf{head}(ce))$, and tail $\Psi(\mathbf{tail}(ce))$.
2. For each path pp , check whether $pp \models ce$ using the method described in Subsection 4.2.1.
If no such path is found -exit with answer NO.
3. For all paths that satisfy ce , check for common delay. If true - exit with YES. Otherwise, exit with NO.

To check satisfiability of cse , we can extend any algorithm for testing satisfiability of plan definitions with a test of Algorithm 4.2, for any plan assignment for which the plan definition holds.

5 Using Plan Structures

5.1 Maintaining and Reasoning with a Plan Structures Knowledge Base

5.1.1 Classification

Subsumption as a Basic Reasoning Means

5.2 Retrieval

5.3 Planning with Plan Structures

6 Conclusion

that are intended to specify that *getFlour* is always *shared*. The formal handling requires the definition of *ground instances* of C-expressions. *Satisfiability* of C-expressions is defined as satisfiability of all ground instances. Similar extensions appear in [6].

4.3 Plan Schemas

A *plan schema* is a pair (def, cse) of a plan definition and a CS-expression, such that the lefthand side of the definition is $\mathbf{head}(ce)$, for every C-expression ce in cse . A plan schema is satisfied in a semantic structure (\mathcal{P}, Ψ) , if the plan definition and the CS-expression are satisfied in that semantic structure. A plan schemas' knowledge base Δ is a collection of plan schemas, such that the plan definitions alone form a plan definitions' knowledge base. Δ is satisfied in a semantic structure (\mathcal{P}, Ψ) , if all plan schemas in it are satisfied. Δ is satisfiable if it is satisfied in some semantic structure.

Checking Satisfiability

The question whether a given knowledge base of plan schemas is satisfiable is important, since it means that the given collection of plan definitions indeed describes a “real” collection of structured plans, and the associated CS-expressions specify consistent SCS-s in these structured plans. By Theorem 4.2 we know that if the Plan Structures Language is restricted to include no compound plan terms, then every knowledge base of plan definitions is satisfiable. Moreover, the proofs of Theorems 4.1 and 4.2, imply also an algorithm for construction of (structured) plans that satisfy the plan definitions in the knowledge base (linear in the size of the plan definitions).

We now turn our attention to the CS-expression

$$cse = ce_1 \oplus ce_2 \oplus \dots \oplus ce_n$$

that is associated with a plan definition

$$\langle name\ term \rangle := \langle plan\ term \rangle.$$

By definition of plan schemas: $\mathbf{head}(ce_i) = \langle name\ term \rangle$, for all $1 \leq i \leq n$. Hence, given a semantic structure (\mathcal{P}, Ψ) , $\tilde{\Psi}(ce_i)$, ($1 \leq i \leq n$), is a sharing constraint in $\Psi(\langle name\ term \rangle)$. In order to check whether ce_i is satisfied in (\mathcal{P}, Ψ) , we need to check whether $\tilde{\Psi}(ce_i)$ is not empty, and is a consistent sharing constraint

Definition 4.4 A C-expression ce is satisfied in a semantic structure (\mathcal{P}, Ψ) (with notation $(\mathcal{P}, \Psi) \models ce$), if $\tilde{\Psi}(ce) \neq \emptyset$, and is a consistent sharing constraint in $\Psi(\text{head}(ce))$. ce is satisfiable if it is satisfied in some semantic structure.

Proposition 4.7 For a positive C-expression ce , $(\mathcal{P}, \Psi) \models ce$, if and only if all plan-paths in $\tilde{\Psi}(ce)$ have a common delay.

Proof: Immediate from Theorem 3.3. \square

4.2.2 CS-expressions

Definition 4.5 A CS-expression has the form

$$cse = ce_1 \oplus ce_2 \oplus \dots \oplus ce_n \quad (n \geq 1)$$

where $ce_i (1 \leq i \leq n)$ is a C-expression. It denotes a sharing-constraints-system:

$$\tilde{\Psi}(cse) = \{ \tilde{\Psi}(ce_i) \mid 1 \leq i \leq n \}$$

A CS-expression is *satisfied in a semantic structure* (\mathcal{P}, Ψ) , if all C-expressions in it are satisfied in that semantics structure. It is *satisfiable* if it is satisfied in some semantic structure.

Example 10 A CS-expression for the baking example:

$$cs = \text{bakeCheeseAppleCakes}, * \text{takeOutFlour} \oplus \neg \text{bakeCheeseAppleCakes}, * \text{beatYolks}$$

$$\tilde{\Psi}(cs) = \{ \{ \langle a_2, a_6, a_{11} \rangle, \langle a_1, a_3, a_9 \rangle \}, - \{ \langle a_1, a_4, a_{10} \rangle, \langle a_2, a_7, a_{12} \rangle \} \}$$

\square

4.2.3 Extending the Constraint Language

The constraints language can be further extended by allowing variables as selectors. We can get expressions like:

$$X, * \text{getFlour}$$

that all conditions hold. This algorithm calls Algorithm 4.1, for testing the third condition in Definition 4.3. Efficiency can be gained by starting the selection from $\mathbf{tail}(pp)$.

The interpretation of *C-expressions* is defined inductively. The mapping $\tilde{\Psi}$ is extended to C-expressions in the following way:

1. For pe , a path-expression: $\tilde{\Psi}(pe) = \{pp \mid pp \models pe\}$
2. For C-expressions ce_1, ce_2 :
 - $\tilde{\Psi}(ce_1 + ce_2) = \begin{cases} \tilde{\Psi}(ce_1) \cup \tilde{\Psi}(ce_2) & \Psi(\mathbf{head}(ce_1)) = \Psi(\mathbf{head}(ce_2)), \Psi(\mathbf{tail}(ce_1)) = \Psi(\mathbf{tail}(ce_2)) \\ \emptyset & \textit{otherwise} \end{cases}$
 - $\tilde{\Psi}(ce_1 \cdot ce_2) =^4 \begin{cases} \tilde{\Psi}(ce_1) \cdot \tilde{\Psi}(ce_2) & \Psi(\mathbf{tail}(ce_1)) = \Psi(\mathbf{head}(ce_2)) \\ \emptyset & \textit{otherwise} \end{cases}$
 - $\tilde{\Psi}(\neg ce_1) = -\tilde{\Psi}(ce_1)$

Proposition 4.6 *For a C-expression ce , and a semantic structure (\mathcal{P}, Ψ) , $\tilde{\Psi}(ce)$ is a sharing constraint in $\Psi(\mathbf{head}(ce))$.*

Example 9 *Plan-paths satisfying C-expressions from Example 7:*

1. $\tilde{\Psi}((bakeCheeseAppleCakes, *preIngApple \cdot preIngApple, *takeOutFlour)) = \{ \langle a_2, a_6 \rangle \} \cdot \{ \langle a_{11} \rangle \} = \{ \langle a_2, a_6, a_{11} \rangle \} = \tilde{\Psi}(bakeCheeseAppleCakes, *preIngApple, *takeOutFlour)$
2. $\tilde{\Psi}((bakeCheeseAppleCakes, *preIngApple, *takeOutFlour + bakeCheeseAppleCakes, *preIngCheese, *takeOutFlour)) = \{ \langle a_2, a_6, a_{11} \rangle \} \cup \{ \langle a_1, a_3, a_9 \rangle \} = \{ \langle a_2, a_6, a_{11} \rangle, \langle a_1, a_3, a_9 \rangle \} = \tilde{\Psi}(bakeCheeseAppleCakes, *takeOutFlour)$
3. $\tilde{\Psi}(\neg bakeCheeseAppleCakes, *beatYolks) = - \{ \langle a_1, a_4, a_{10} \rangle, \langle a_2, a_7, a_{12} \rangle \}$

□

⁴The \cdot on the left side is syntactical; the \cdot on the right side is the concatenation operation between sets of plan paths. It evaluates to the set of plan paths which are the concatenations of plan paths in the first argument with plan paths in the second argument.

Definition 4.3 A plan path $pp = \alpha_0, \alpha_1, \dots, \alpha_n$ ($n \geq 0$), where $\alpha_i = (p_i, id_i, p_{i+1}, delay_i)$, satisfies a path expression $pe = sel_0\{ \langle timeTerm \rangle_0 \}, sel_1\{ \langle timeTerm \rangle_1 \}, \dots, sel_{m+1}$ ($m \leq n$)

(with notation $pp \models pe$), if and only if there exist nodes $p_{j_0}, p_{j_1}, \dots, p_{j_{m+1}}$, along the path pp , such that:

- $p_{j_0} = \mathbf{head}(pp)$, $p_{j_{m+1}} = \mathbf{tail}(pp)$
- For all $0 \leq i \leq m + 1$, if sel_i is $\langle plan\ term \rangle$, then $j_i = j_{i-1} + 1$.
- If pt_i is the plan term within sel_i ($sel_i = pt_i$ or $sel_i = *pt_i$), then $p_{j_i} \in \tilde{\Psi}(pt_i)$.
- For $0 \leq i \leq m$, if $timeTerm_i$ is specified, then $\Psi(timeTerm_i) = delay(\langle \alpha_{j_i}, \alpha_{j_i+1}, \dots, \alpha_{j_{i+1}} \rangle)$.

Example 8 Plan-paths in Figure 3 that satisfy path expressions from Example 7:

The plan paths $\langle a_1, a_3, a_9 \rangle$ and $\langle a_2, a_6, a_{11} \rangle$ satisfy the path expression

$$bakeCheeseAppleCakes, * takeOutFlour.$$

Only $\langle a_2, a_6, a_{11} \rangle$ satisfies the path expression

$$bakeCheeseAppleCakes, * preIngApple, * takeOutFlour.$$

Neither path satisfies the path expression

$$bakeCheeseAppleCakes . \mathbf{duration}(preIngCheese), * preIngApple, * takeOutFlour.$$

□

Proposition 4.5 For a path expression pe , and a semantic structure (\mathcal{P}, Ψ) , the plan paths that satisfy pe , are plan-paths-in- $\Psi(\mathbf{head}(pe))$, and they form a sharing constraint in p .

Note, of course, that the set of plan paths that satisfy pe may be empty. This is the case if pe specifies no plan-path-in- $\Psi(\mathbf{head}(pe))$. In any case, $\Psi(\mathbf{head}(pe))$ is always a single plan, since $\mathbf{head}(pe)$ is restricted to be a name term.

Given a path expression pe as in Definition 4.3, and pp a plan-path-in- $\Psi(\mathbf{head}(pe))$, the question “does pp satisfy pe ?” is relevant for the more general satisfaction of C-expressions, below. The question can be answered by an algorithm that systematically selects the nodes p_{j_i} , ($0 \leq i \leq m + 1$) on pp , and verifies

This characteristic is important since it guarantees that in a given semantic structure (\mathcal{P}, Ψ) , $\Psi(\mathbf{head}(ce))$ and $\Psi(\mathbf{tail}(ce))$ are plans. This is not the case for arbitrary plan terms that denote sets, possibly empty, of plans.

Example 7 *C-expressions for the baking example:*

Example 1 includes two C-expressions associated with the plan definition for `bakeCheeseAppleCakes`. The C-expression

$$bakeCheeseAppleCakes, * takeOutFlour$$

is a positive C-expression that specifies all paths from `bakeCheeseAppleCakes` to `takeOutFlour`. The C-expression

$$\neg bakeCheeseAppleCakes, * beatYolks$$

is a negative C-expression that specifies all paths from `bakeCheeseAppleCakes` to `beatYolks`. The C-expression

$$bakeCheeseAppleCakes, * preIngApple, * takeOutFlour$$

restricts the paths between `bakeCheeseAppleCakes` and `takeOutFlour` to go through `preIngApple`. It can also be written as the “composite” C-expression

$$(bakeCheeseAppleCakes, * preIngApple \cdot preIngApple, * takeOutFlour).$$

The C-expression

$$bakeCheeseAppleCakes \cdot \mathbf{duration}(preIngCheese), * preIngApple, * takeOutFlour$$

further restricts the path portion between `bakeCheeseAppleCakes` and `preIngApple` to have delay $\mathbf{duration}(preIngCheese)$.

□

Semantics:

Let (\mathcal{P}, Ψ) be a semantic structure for the Plans Structures language.

4.2.1 C-Expressions

Syntax:

Definition 4.1 A path expression is an expression, pe , of the form:

$$pe = sel_0 \{ \langle timeTerm \rangle_0 \}, sel_1 \{ \langle timeTerm \rangle_1 \}, \dots, sel_{m+1} \quad (m \geq 0)$$

where sel_i is either $\langle plan\ term \rangle$, or $*\langle plan\ term \rangle$. sel_0, sel_{m+1} are name terms. $\mathbf{head}(pe) = sel_0$, $\mathbf{tail}(pe) = sel_{m+1}$.

The selectors in the path expression denote nodes in the denoted plan paths. If sel_i is $\langle plan\ term \rangle$, then there are single arcs between the nodes denoted by sel_{i-1} and sel_i . If sel_i is $*\langle plan\ term \rangle$, then there are any length paths between the nodes denoted by sel_{i-1} and sel_i . The time terms are optional. If $\langle timeTerm \rangle_{i-1}$ is specified, then it specifies the delay of paths between the nodes denoted by sel_{i-1} and sel_i . Hence, if the path expression is only sel_0, sel_1 , then it denotes all single arcs between the nodes denoted by the selectors. If the path expression is $sel_0, * sel_1$, then it denotes all plan paths between the nodes denoted by the selectors. If the path expression is $sel_0, 5, * sel_1$, then it denotes all length 5 plan paths between the nodes denoted by the selectors.

Definition 4.2 C-expressions:

1. A path expression is a positive C-expression.
2. Let ce_1, ce_2 , be positive C-expressions. Then the following are also positive C-expressions:
 - $ce = (ce_1 + ce_2)$. In that case: $\mathbf{head}(ce) = \mathbf{head}(ce_1)$, $\mathbf{tail}(ce) = \mathbf{tail}(ce_1)$.
 - $ce = (ce_1 \cdot ce_2)$. In that case: $\mathbf{head}(ce) = \mathbf{head}(ce_1)$, $\mathbf{tail}(ce) = \mathbf{tail}(ce_2)$.
3. For ce , a positive C-expression, $\neg ce$ is a negative C-expression. In that case: $\mathbf{head}(\neg ce) = \mathbf{head}(ce)$, $\mathbf{tail}(\neg ce) = \mathbf{tail}(ce)$.

Below, we omit the qualification "positive" from C-expressions, so that we have "plain" and negative C-expressions.

Property 4.4 For a C-expression ce , $\mathbf{head}(ce)$ and $\mathbf{tail}(ce)$ are name terms.

• **Method:**

1. Construct a syntax graph, $G(p\text{term})$ for $p\text{term}$.
2. In a bottom up manner, starting from the leaves of $G(p\text{term})$, identify nodes n in $G(p\text{term})$ with nodes $Id(n)$ in $G(p)$, such that $Id(n) \in \tilde{\Psi}(n)$. This is possible for the following reasons:
 - The leaves of $G(p\text{term})$ are elementary plan terms, and they should be identified with “themselves”, as elementary plans in \mathcal{H} .
 - For each new level in $G(p\text{term})$, a subterm

$$q\text{term} = \mathbf{constructor}(q\text{term}_1, \dots, q\text{term}_k)$$

is identified with a plan

$$q = ((Id(q\text{term}_1), t_1), \dots, (Id(q\text{term}_k), t_k))$$

in $G(p)$, if the temporal structure of q satisfies the meaning of the plan constructor **constructor**, i.e., $q \in \tilde{\Psi}(q\text{term})$. This is always a local condition, that involves just the elements of q . In general, there can be several ways to identify $q\text{term}$ with a plan in $G(p)$.

If no right identification is possible – backtrack to previous level.

If no backtrack is possible – exit with answer NO.

3. If step (2) is successfully completed – exit with answer YES.

Complexity analysis: The algorithm requires at least a full scan of $G(p\text{term})$. In worst case, all possible matchings of $G(p\text{term})$ with subgraphs of $G(p)$ will be checked. Hence, the worst case analysis is ...

4.2 The Constraints Language

The constraints language includes *Constraint (C)-expressions*, and *Constraint-System (CS)-expressions*. C-expressions are (regular) expressions that denote sets of plan paths, each with a common head and a common tail, i.e., sharing constraints. The specification provided in a C-expression can be very detailed, so that the C-expression denotes a single plan path. The specification can be partial, and in that case the C-expression denotes all plan paths that satisfy the partial specification.

Proof: (Sketch)

Given a KB Δ , do the following transformations on operators:

1. Replace operators **and**, **partial_and**, **before**, **temporal_disjoint**, **starts_before**, **ends_after** by the operator **sequential**.
2. Replace operators **after**, **starts_after**, **ends_after** by the operator **sequential**, and exchange the order of their arguments (these are binary operators).
3. Replace the operator **included** by the operator **simultaneous**.

Denote the new KB Δ' .

Lemma 4.3 Δ is satisfiable if Δ' is.

Proof: For a plan term pt , let pt' be the plan term after the above transformation. Then, for all transformations above we have: $\tilde{\Psi}(pt') \subseteq \tilde{\Psi}(pt)$. \square

The theorem follows from Theorem 4.1, and the above Lemma. \square

We conjecture that the general satisfiability problem of plan definitions KBs can be reduced to satisfiability over “Herbrand style” semantic structures \mathcal{H} , as in the proof of theorem 4.1. Then, in principle, given a KB δ , we can systematically generate plans in \mathcal{H} , and check whether they satisfy Δ . The problem is, of course, that if the righthand sides of plan definitions are partial plan terms, there can be unlimited number of plans in their extensions (under $\tilde{\Psi}$). The full handling of this problem requires careful investigation of the impact of different partial plan constructors, and of the structure of compound plan terms. This is out of the scope of this paper.

Nevertheless, it seems that the ability to check whether a given plan in \mathcal{H} satisfies a plan term is essential. That is, given $p \in \mathcal{H}$, and a plan term $pterm$, the problem is to determine whether $p \in \tilde{\Psi}(pterm)$ or not. We sketch here an algorithms for non-compound plan terms. It is based on the idea that the syntax graph of a non-compound $pterm$ should be isomorphic in structure to a subgraph of p . This is only a sketch, as an exact algorithm depends on the set of plan constructors.

Algorithm 4.1 • **input:** $p \in \mathcal{H}$, $pterm$ a non-compound plan term.

- **Output:** YES – if $p \in \tilde{\Psi}(pterm)$, and NO – otherwise.

Satisfiability:

A definition $\langle name\ term \rangle := \langle plan\ term \rangle$ holds in a semantic structure *iff*

$$\Psi(\langle name\ term \rangle) \in \tilde{\Psi}\langle plan\ term \rangle.$$

A plan definitions KB Δ is *satisfied* in a semantic structure, $(\mathcal{P}, \Psi) \models \Delta$, if all definitions are satisfied.

Δ is *satisfiable* if it is satisfied in some semantic structure.

Theorem 4.1 *If the Plan Structures Language is restricted as follows:*

1. *Constructors: Only the four main complete constructors; no partial constructors.*
2. *No compound plan terms.*

then every knowledge base is satisfiable.

Proof: (Sketch)

Construct syntax graphs for all plan terms in a given KB Δ , and associate the left sides of definitions with the roots of their right sides. Note that there are no cycles. In a bottom up manner, do the following:

1. Identify time point symbols, time arithmetic function symbols, and time operator symbols, with time points, time arithmetic functions, and time operators, respectively. Compute all time terms, and replace them by the result of their evaluations.
2. Replace **simultaneous** and **sequential** nodes by **temporal.concat** nodes, by computing the exact delays of the arguments.

This process is possible since all plan terms are complete, and the syntax graphs are acyclic. The resulting graphs are (structured) plans in a “Herbrand style” semantic structure (\mathcal{H}, Ψ) , where

$$\mathcal{H} = (object\ symbols, action\ terms, action\ operator\ symbols, \langle time\ points, + \rangle, EP, P).$$

The obtained structured plans satisfy the definitions in Δ . \square

Theorem 4.2 *If the Plan Structures Language includes no compound plan terms, then every knowledge base is satisfiable.*

- Complete constructor: A complete constructor symbol stands for a specific operator on plans. The plan structure, in that case, denotes the set of all structured plans that result from application of the operator to plans denoted by the argument terms. For example, the constructor symbol **sequential** denotes the operator **sequential**, which combines plans in sequence. Hence, for $n \geq 2$:

$$\tilde{\Psi}(\mathbf{sequential}(\langle pterm_1 \rangle, \dots, \langle pterm_n \rangle)) = \{ \mathbf{sequential}(p_1, \dots, p_n) \mid p_i \in \tilde{\Psi}(\langle pterm_i \rangle), 1 \leq i \leq n \}$$

- Partial constructor: A partial constructor denotes a constraint on plans. For example, the partial constructor **before** denotes the constraint $C_{\mathbf{before}}$, defined as follows:

$C_{\mathbf{before}}(p, q, r)$ is true *iff*

- p is a structured plan;
- p has at least two elements $(q, t_1), (r, t_2)$, in any order;
- $t_1 + \mathbf{end}(q) \leq t_2 + \mathbf{start}(r)$

That is, $C_{\mathbf{before}}(p, q, r)$ characterizes structured plans with at least two direct components, such that one ends before the other starts. A plan structure formed with the **before** constructor denotes structured plans that satisfy the $C_{\mathbf{before}}$ constraint with plans denoted by arguments of the plan structure term. That is:

$$\tilde{\Psi}(\mathbf{before}(\langle pterm_1 \rangle, \langle pterm_2 \rangle)) = \{ p \mid C_{\mathbf{before}}(p, q, r) \text{ is true for } \\ q \in \tilde{\Psi}(\langle pterm_1 \rangle), r \in \tilde{\Psi}(\langle pterm_2 \rangle) \}$$

In general, for a partial constructor **parCon**, that denotes a constraint $C_{\mathbf{parCon}}$:

$$\tilde{\Psi}(\mathbf{parCon}(\langle pterm_1 \rangle, \dots, \langle pterm_n \rangle)) = \{ p \mid C_{\mathbf{parCon}}(p, p_1, \dots, p_n) \text{ is true for } \\ p_i \in \tilde{\Psi}(\langle pterm_i \rangle), 1 \leq i \leq n \}$$

The partial constructors are given their standard meanings, as in [1, 5].

4. Compound plan terms:

$$\tilde{\Psi}(\langle \langle pterm_1 \rangle, \langle pterm_2 \rangle \rangle) = \tilde{\Psi}(\langle pterm_1 \rangle) \cap \tilde{\Psi}(\langle pterm_2 \rangle)$$

$$\tilde{\Psi}(\langle \langle pterm_1 \rangle; \langle pterm_2 \rangle \rangle) = \tilde{\Psi}(\langle pterm_1 \rangle) \cup \tilde{\Psi}(\langle pterm_2 \rangle)$$

- ($\langle plan\ term \rangle, \langle plan\ term \rangle$).
- ($\langle plan\ term \rangle; \langle plan\ term \rangle$).

Definitions: $\langle name\ term \rangle := \langle plan\ term \rangle$.

Both sides of a definition should be, together, elementary or not. If the right side is partial, then the definition is as such. Otherwise, the definition is complete.

Plan Definitions' Knowledge Base (KB): A set of definitions, with no two definitions having the same left side, and no “name cycles”³.

Semantics

A semantic structure for the Plan Structures language is a pair (\mathcal{P}, Ψ) , where \mathcal{P} is a plan space, and Ψ is a partial interpretation mapping that associates symbols with their natural semantic counterparts over \mathcal{P} . Elementary plan names are associated with elementary plans, and structured plan names are associated with structured plans. Ψ is undefined for plan operator symbols, and for *NIL*.

Extending the Interpretation Mapping Ψ :

Ψ is extended to all terms, except for plan terms, in the standard way. In particular, time terms are associated with time points, duration terms with durations, and action terms are associated with actions. The extension of Ψ to plan terms is called $\tilde{\Psi}$. It associates a plan term with a **set of plans**. It is defined as follows:

1. **Name terms:** $\tilde{\Psi}(\langle name\ term \rangle) = \{\Psi(\langle name\ term \rangle)\}$

2. **Elementary plan terms:**

$$\tilde{\Psi}([\langle action\ term \rangle, \langle duration\ term \rangle]) = \{ (\Psi(\langle action\ term \rangle), \Psi(\langle duration\ term \rangle)) \}$$

$$\tilde{\Psi}([\langle action\ term \rangle, NIL]) = \{ ep \mid ep \in EP, \mathbf{action}(ep) = \Psi(\langle action\ term \rangle) \}$$

3. **Plan structures:** $\langle constructor\ symbol \rangle(term_1, \dots, term_n)$.

³Name cycles can be formally defined by considering plan terms as *syntaxgraphs*, and associating the left side of each definition with the root of the right side's graph. The knowledge base has name cycles if the syntax graphs include cycles.

The exact set of plan operators is, probably, domain dependent. Different sets yield different languages.

Terms: The terms of the language are regular well typed first order terms. They can be defined recursively, starting from all non operator/function symbols, using well typed applications of operator/function symbols to terms. The main types of terms are listed below:

1. **Time terms:**

- time point symbols.
- $\langle \text{time arithmetic function symbol} \rangle (\langle \text{Time term} \rangle s)$.
- $\langle \text{time operator symbol} \rangle (\text{Appropriately typed terms})$.

Certain time terms, like the ones generated by the **duration** operator, are qualified as *duration* terms.

2. **Action terms:** $\langle \text{action operator symbol} \rangle (\langle \text{object symbol} \rangle s)$.

3. **Plan terms:**

- **Name terms:**
 - Elementary: Elementary plan names.
 - Non-Elementary: Structured plan names.
- **Elementary plan terms:**
 - $\langle \text{elementary name term} \rangle$.
 - $[\langle \text{action term} \rangle, \langle \text{duration term} \rangle]$.
 - $[\langle \text{action term} \rangle, \text{NIL}]$.

In the latter case, the term is *partial*.

- **Plan structures:**
 - $\langle \text{non-elementary name term} \rangle$.
 - $\langle \text{constructor symbol} \rangle (\langle \text{term} \rangle s)$.

If the constructor is partial, or one of the argument terms is partial, then the whole term is *partial*. Otherwise, it is *complete*.

- **Compound plan structures:**

4 A Plan Structures Language

In the Plan Structures approach, a plan schema consists of two parts: A *plan definition*, and a *Constraint-System expression (CS-expression)*. Plan definitions use a small set of *plan operators* whose meanings are built into the semantics. Different sets of plan operators give rise to different languages for plan definitions. The semantics is set theoretic, since a plan definition may provide a partial specification of plans (reminds description languages [2, 10, 7, 9, 11, 8]). The CS expressions describe sets of plan-paths. They borrow ideas for path specification from the XSQL query language of [6].

4.1 Plan Definitions

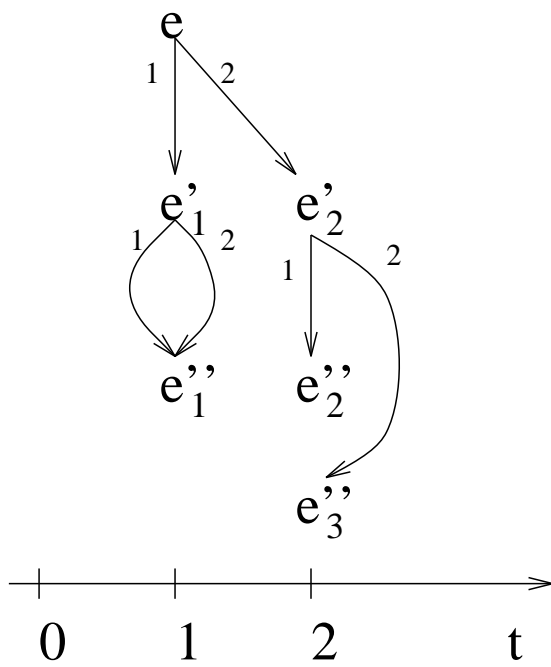
Plan definitions associate names with plan terms that specify plans. The specification can be *complete* or *partial*. A complete specification denotes a single plan in a given plan space. A partial specification denotes a set of plans. Examples of plan definitions appear in Example 1. The version defined below does not support plan arguments and object definitions. The extension is straightforward.

Syntax

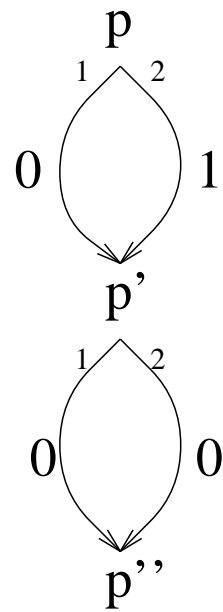
A Plan Definitions' language includes symbols for the following types: Time points, action operators, objects, numbers, elementary and structured plan names, time arithmetic functions, and plan operators. *NIL* is a special symbol. All function and operator symbols have associated arities.

Plan operator symbols can be further classified, as follows:

- Complete constructors, e.g., [], **temporal_concat**, **sequential**, **simultaneous**.
- Selectors, e.g., **remove**, **direct_components**, **non_empty_time_point**.
- Time operators, e.g.,
 - Self operators, e.g., **duration**, **start**, **end**, **interval**.
 - Relative operators, e.g., **delays**, **relative_starts**, **relative_ends**, **relative_intervals**.
- Partial constructors, e.g., **and**, **partial_and**, **before**, **after**, **temporal_disjoint**, **starts_before**, **starts_after**, **ends_before**, **ends_after**, **included**.



a) structured event



b) structured plan

Figure 4: A structured event and a corresponding structured plan

5. *Otherwise, answer “no”.*

The algorithm takes time on the order of the number of arcs in the graph of e , since it is simply a breadth-first search algorithm with constant-time additional operations on each arc. Checking the sharing constraints is also easy (polynomial time), after running the above algorithm.

Let SC be a sharing constraint in p with source p_s and sink p_e . Find the set of nodes $E_s = \{e_s \mid \Lambda(e_s) = p_s\}$ (this takes linear time, but could also have been done during the run of the above algorithm). For each $e_s \in E_s$, and each $q \in SC$, there is exactly one path u with the same index string as q . Let $f(e_s, q) = \text{sink}(u)$, for the u defined above.

If SC is a positive sharing constraint, then it is satisfied just when for every $e_s \in E_s$, there exists a node $e_e = f(e_s, q)$ that is the same node for every $q \in SC$. If SC is a negative sharing constraint, then it is satisfied just when for every $e_s \in E_s$, if $q_1, q_2 \in SC$ then $q_1 \neq q_2$ implies $f(e_s, q_1) \neq f(e_s, q_2)$.

Finding all the end nodes requires time $O(|SC| * |E_s| * l)$, where l is the length of the longest path in SC . Checking whether the nodes are the same (positive constraint) requires a further $O(|SC| * |E_s|)$. Checking whether they are all different requires a further $O(|SC|^2 * |E_s|)$. All in all, checking whether a positive constraint is satisfied takes $O(|SC| * |E_s| * l)$, and checking a negative constraint takes $O(|SC| * |E_s| * (l + |SC|))$. $|E_s|$ may be exponential in l , but since it is less than the size of the graph of e , which is an element of the input, that is not a serious difficulty.

p may be translated into an exponential number of paths in the event space (as is the case in the proof of ??).

Another important issue is checking whether a given structured event is a model for a plan (whether with or without sharing constraints). We present a simple polynomial-time algorithm for answering this question when there are no constraints, and then discuss a simple method for checking whether constraints are satisfied.

The basic algorithm scans the components of e in a breadth-first search and sets values of Λ to components of e . These are attached to the relevant event nodes for efficiency. A queue (initially empty) is used to hold $(even, plan, time)$ triplets for processing.

Algorithm 3.1 • *Input: a structured plan p (with start time of all components zero), a structured event e , both in graph form.*

• *Output: the mapping Λ from components of e to components of p if it exists (i.e. e is a model of p), and answer “no” otherwise.*

1. *Process the triplet (e, p, X) , where X is a special “dont’ care” value.*
2. *If the queue is empty, return. Otherwise, remove an item (e', p', t) from the head of the queue and process it.*

Processing an item (e', p', t) works as follows:

1. *If $start(e') \neq t$ and $t \neq X$, answer “no”.*
2. *Otherwise, if $t = X$ then set $t = start(e')$.*
3. *If e', p' are both elementary, then check that they have equal action and duration, and if they do then set $\Lambda(e') = p$. Otherwise, answer “no”.*
4. *Otherwise, if e' and p' are both structured, then do the following steps:*
 - (a) *Let n be the number of elements of e' . If the number of elements of p' is not n , answer “no”.*
 - (b) *For each $1 \leq i \leq n$, insert $(e^i, p^i, t + t_i)$ into the tail of the queue, where (p', i, p^i, t_i) is the i th element (arc) of p' , and (e', i, e^i) is the i th element (arc) of e' .*

(p^2, t_i) , such that $p^2 = \Lambda(e^2)$. The arc in $G(p)$ that corresponds to the arc (e^1, i, e^2) is defined as follows:

$$\Lambda((e^1, i, e^2)) = (\Lambda(e^1), i, \Lambda(e^2), t_i)$$

Likewise, for a path u in $G(e)$, $u = ((e_1, i_1, e_2), \dots, (e_n, i_n, e_{n+1}))$, we define its corresponding path as:

$$q = \Lambda(u) = (\Lambda((e_1, i_1, e_2)), \dots, (\Lambda((e_n, i_n, e_{n+1}))))$$

The delay of path u is defined as $delay(u) = start(e_{n+1}) - start(e_1)$. The delay of q is the sum of all arc delays, and can easily be shown to be equal to the delay of u . We call the string i_1, i_2, \dots, i_n the *index string* of u (or q). By construction, u and $\Lambda(u)$ have the same index string.

3.2.2 Restricting the semantics of Plans by Considering Sharing Constraint Systems

Let \mathcal{P} be a plan space with a set of plans P , with (\mathcal{E}, Γ) a semantic structure for \mathcal{P} , where the set of events in \mathcal{E} is E . For the following definitions, let p be a structured plan in P , SC_p be a sharing constraint in p (positive or negative), and $e \in \Gamma(p, t)$ for some time point t . Let p_s be the source plan of the constraint, and e_s be some source event of the constraint (i.e. $\Lambda(e) = p_s$), and U_{e_s} be the set of all paths with source e_s with an image in the constraint:

$$U_{e_s} = \{u \mid source(u) = e_s \wedge \Lambda(u) \in SC_p\}$$

Definition 3.4 *e satisfies the positive sharing constraint SC_p if and only if all paths in U have a common tail (sink). Likewise, e satisfies the negative sharing constraint SC_p if and only if all paths in U have distinct tails (sinks).*

Definition 3.5 *e satisfies a sharing constraint system in p SCS_p just when it satisfies all sharing constraints in SCS_p . SCS_p is satisfied in a semantic structure (\mathcal{E}, Γ) for \mathcal{P} , if some event in \mathcal{E} satisfies it. SCS_p is consistent if it is satisfied in some semantic structure.*

Theorem 3.3 *Let SCS_p be a positive constraint system in p . Then SCS_p is consistent iff it is a proper positive constraint system.*

See appendix ?? for a proof of this theorem. For constraint systems with negation, there is no such easy criterion for checking consistency. In fact, we believe that given a plan p and a sharing constraint system, the problem of checking consistency is hard. The reason for that is that a set of paths U in the plan description

Proof:

START*****

The proof is too condensed, for me, and I assume for most readers. Perhaps, associating it with a figure, may solve all misunderstandings.

END*****

By construction: we show a structured plan with n components, and show that any arbitrary model for it has 2^n elementary events. Consider the following structured plan:

- Elementary plan p_0 , with $\mathbf{duration}(p_0) = 1$, $\mathbf{start}(p_0) = 0$.
- For $n \geq i > 0$ we have a structured plan $p_i = ((p_{i-1}, 0), (p_{i-1}, 2^{i-1}))$, (with $\mathbf{start}(p_0) = 0$, $\mathbf{duration}(p_i) = 2^i$).

Claim: let $e \in \Gamma(p_n, t)$ for some t . Then e contains 2^n elementary events e_{0_k} with $0 \leq k < 2^n$, such that $\mathbf{start}(e_{0_k}) = t + k$. Naturally, since all the elementary events have a different start time, they are disjoint.

Proof of claim: by induction on i . Base case: $i = 0$. Let $e_0 \in \Gamma(p_0, t)$ for some t . But that implies that $\mathbf{start}(e_0) = t$, and naturally includes 1 elementary event (itself). Now, let the claim hold for p_{i-1} , and show that the claim holds for p_i . From the definition of Λ :

$$p_i = ((p_{i-1}, 0), (p_{i-1}, 2^{i-1})) = \Lambda(e_i) = ((\Lambda(e_{i-1}^0), 0), (\Lambda(e_{i-1}^1), 2^{i-1}))$$

For some e_{i-1}^0, e_{i-1}^1 , with $\mathbf{start}(e_{i-1}^0) = t$ and $\mathbf{start}(e_{i-1}^1) = t + 2^{i-1}$, which may or may not be disjoint. However, by the induction hypothesis, e_{i-1}^0 contains elementary events $e_{0_k}^0$ for $0 \leq k < 2^{i-1}$ such that $\mathbf{start}(e_{0_k}^0) = t + k$ (since $p_{i-1} = \Lambda(e_{i-1}^0)$, where $\mathbf{start}(e_{i-1}^0) = t$). Likewise, since $p_{i-1} = \Lambda(e_{i-1}^1)$ (with $\mathbf{start}(e_{i-1}^1) = t + 2^{i-1}$), then e_{i-1}^1 contains 2^{i-1} elementary events $e_{0_k}^1$ for $0 \leq k < 2^{i-1}$, such that $\mathbf{start}(e_{0_k}^1) = t + 2^{i-1} + k$. In principle, the events $e_{0_k}^0$ may overlap the events $e_{0_k}^1$. In this case, however, there are clearly 2^i start times, ranging from t to $t + 2^{i-1} + 2^{i-1} - 1 = t + 2^i - 1$, thereby proving the claim. \square

Semantics of paths

The semantics of a plan p as a set of events induces a mapping between arcs in $G(e)$ and arcs in $G(p)$. Let (e^1, i, e^2) be an arc in $G(e)$, and let $p^1 = \Lambda(e^1)$. Then, for some time point t_i , the i -th element of p^1 is

3. If p and all its components have a start of 0, then there exists an event e (in some domain E) such that $\Lambda(e) = p$.
4. $\mathbf{duration}(\Lambda(e)) = \mathbf{duration}(e)$

The semantics of a structured plan is an inverse of Λ :

Definition 3.3 A semantic structure for a plan space $\mathcal{P} = (OBJ, A, OP, < T, + >, EP, P)$ is a pair (\mathcal{E}, Γ) , where \mathcal{E} is an events' space $(A, < T, + >, EE, E)$, and Γ is the inverse mapping of the above defined Λ :

$$\Gamma(p, t) = \{e \mid e \in E, p = \Lambda(e), \mathbf{start}(e) = t\}$$

By Claim 3.1, if E is complete and $p \in P$ satisfies $\mathbf{start}(p) = 0$, then $\Gamma(p, t) \neq \emptyset$, for all $t \in T$. The meaning of plans can be extended to all plans, by partitioning the set of plans P into equivalence classes, such that in each class there is a single plan p with $\mathbf{start}(p) = 0$. This plan can be taken as the *representative* of its class. The meaning of all plans in a class is taken as the meaning of the class representative. The partition of P can be obtained by an operation *shift2Zero*, defined as follows: For an elementary plan ep , $shift2Zero(ep) = ep$; for a structured plan $p = ((p^1, t_1), (p^2, t_2), \dots, (p^n, t_n))$, with $\mathbf{start}(p) = t$, $shift2Zero(p) = ((p^1, t_1 - t), (p^2, t_2 - t), \dots, (p^n, t_n - t))$. It is easy to verify that for any plan p , $\mathbf{start}(shift2Zero(p)) = 0$. The partitioning of P is obtained by relating plans that share a common *shift2Zero* form.

START*****

Again, an example, that relates to the recent example is needed.

Example 6

□

END*****

3.2.1 Properties of Semantic Structures for Plans

In the following theorems, we assume that a semantic structure is given. .

Structured plans can be used to represent a large number of events in a compact form.

Theorem 3.2 *There exist structured plans for which every denotation contains an exponential number of elementary events.*

Theorem about

larity classes is

ted

3.2 Semantics of Plans

The meaning of a plan is given in terms of a set of events that preserve the structure of the plan. Repetition of plan components is unfolded into several events, one for each repetition, while sharing of component plans is captured by mapping a shared component into a single event. The semantics of a plan is given with respect to a reference time point. First, we define the meaning of plans alone, and then add *SCSs* as restrictions imposed on the original meanings.

We begin by defining when a structured event is a model for a structured plan. Since each plan maps to a multitude of events, we find it more natural to start in the inverse direction, and define a function Λ , mapping from events to plans, i.e.,

$$\Lambda : E \rightarrow P$$

Let e be a structured event. Define the plan modeling it, $\Lambda(e)$ as follows:

- If $e \in EE$ then:

$$\Lambda(e) = (\mathbf{action}(e), \mathbf{duration}(e))$$

- If $e = \mathbf{Se}((e_1, \dots, e_n), ts, te)$ for some $n \geq 1$, then:

$$\Lambda(e) = p = ((p_1, t_1), \dots, (p_n, t_n))$$

where $t_i = \mathbf{start}(e_i) - \mathbf{start}(e)$, and $p_i = \Lambda(e_i)$, for $1 \leq i \leq n$.

For example, consider the structured event figure 4a, where the start time of each event is given by its horizontal position w.r.t. the time axis. The top of the arcs are annotated by their respective element number, i . Figure 4b is the respective structured plan, where $\Lambda(e''_1) = \Lambda(e''_2) = \Lambda(e''_3) = p''$ (for the elementary events), $\Lambda(e'_1) = \Lambda(e'_2) = p'$, and $\Lambda(e) = p$ for the top level event. Plan arcs are annotated by delays, as well as by element number. Note that for most domains E , e is *not* the only structured event which map to p . Duration and action for events and plans are ignored in this example.

Proposition 3.1

1. Λ is a mapping in: $E \rightarrow P$. That is, event e corresponds to a single plan.
2. For any $e \in E$, $\mathbf{start}(\Lambda(e)) = 0$.

□

Structure Operators

The main structure operators are: **temporal_concatenation**, **sequential**, and **simultaneous**.

- **temporal_concatenation** constructs a structured plan $p = ((q_1, t_1), (q_2, t_2), \dots, (q_n, t_n))$ ($n \geq 1$), from given n elements $(q_1, t_1), (q_2, t_2), \dots, (q_n, t_n)$.
- **sequential** constructs a structured plan p from given sub-plans q_1, q_2, \dots, q_n ($n \geq 2$), such that the delay of q_i is the sum of the durations of the preceding sub-plans, and the first sub-plan starts at time point 0 of p . That is, $p = ((q_1, t_1), (q_2, t_2), \dots, (q_n, t_n))$, such that $t_1 = -(\mathbf{start}(q_1))$, and $t_i = \sum_{j=1}^{i-1} \mathbf{duration}(q_j)$, ($i \geq 2$).
- **simultaneous** constructs a structured plan p from given sub-plans q_1, q_2, \dots, q_n ($n \geq 2$), such that they all starts simultaneously, at time point 0 of p . That is, $p = ((q_1, t_1), (q_2, t_2), \dots, (q_n, t_n))$, such that $t_i = -(\mathbf{start}(q_i))$, ($1 \geq i \leq n$).

3.1.2 Sharing Constraints

Definition 3.2 A sharing constraint in p is a collection SC_p of plan paths in p and a sign (positive or negative), such that all paths in $G(SC_p)$, have a common source and a common sink. A (Sharing) Constraint System in p is a set CS_p of sharing constraints in p , each of which may be a negative constraint. A positive constraint where all paths have equal delay is called a proper positive constraint. A constraint system that contains only positive (or proper) constraints is called a positive (or, respectively, proper) constraint system.

An example of a *SCS* in the structured plan presented in Figure 3 was described in the introduction. Intuitively, a positive sharing constraint means that an instance of a plan must be shared (i.e. be the same event during execution) among all the paths in the constraint, while a negative sharing constraint implies that all the instances must be different events during execution.

bakeCheeseCake via *pp*.

□

3.1.1 Operators

Structured plans have associated *structure* and *time* operators. The structure operators can construct structured plans, or select their components. The time operators specify the temporal properties of plans.

Time Operators

The time operators consist of *self* and *relative* operators. The self operators assign to each structured plan a **duration**, **start**, and **end** time points, and also an **interval** value. The relative operators provide the relative time properties of a plan, which is a component of another structured plan, taken as a context. These include the **delays** of the component, i.e., the displacements of the origin of the component plan from the origin of the context plan, the **relative starts**, **relative ends**, and **relative intervals** of the component plan, with respect to the context plan.

Example 5 *In the structured plan presented in Figure 3:*

$$\begin{aligned}
 \mathbf{start}(bakeCheeseAppleCakes) &= \min\{\mathbf{start}(bakeCheeseCake) + 0, \mathbf{start}(bakeApplePie) + 0\} \\
 \mathbf{end}(bakeCheeseAppleCakes) &= \max\{\mathbf{end}(bakeCheeseCake) + 0, \mathbf{end}(bakeApplePie) + 0\} \\
 \mathbf{duration}(bakeCheeseAppleCakes) &= \mathbf{end}(bakeCheeseAppleCakes) - \mathbf{start}(bakeCheeseAppleCakes) \\
 \mathbf{interval}(bakeCheeseAppleCakes) &= [\mathbf{start}(bakeCheeseAppleCakes), \mathbf{end}(bakeCheeseAppleCakes)] \\
 \\
 \mathbf{delays}(beatYolks, bakeCheeseAppleCakes) &= \{3, 7\} \\
 \mathbf{relative_starts}(beatYolks, bakeCheeseAppleCakes) &= \mathbf{start}(beatYolks) \oplus \{3, 7\} \\
 \mathbf{relative_ends}(beatYolks, bakeCheeseAppleCakes) &= \mathbf{end}(beatYolks) \oplus \{3, 7\} \\
 \mathbf{relative_intervals}(beatYolks, bakeCheeseAppleCakes) &= \mathbf{interval}(beatYolks) \oplus \{3, 7\}
 \end{aligned}$$

\oplus is used here in an overloaded way, for adding a time point to all elements of a set of time points, yielding a new set of time points, and for adding a time interval to all elements of a set of time points, yielding a new set of time intervals.²

²These definitions derive from the formal definition of structured histories and their operators ([4]).

a common sub-plan, then that plan may denote simultaneous similar structured events as well as *shared* structured events. If some sub-plan is not elementary then the structured plan has a non-trivial hierarchical structure. Otherwise, it is *flat*. Elementary plans are considered as *flat plans*, as well. Henceforth we use the term *plan* as an inclusive name for elementary and structured plans.

Plans can be drawn as *directed acyclic graphs (multi)-graphs* (DAGs), where internal nodes represent structured plans, leaves represent elementary plans, and arcs are labeled by their index and their time points, called *delays*. Each arc corresponds to an element of the structured plan, and we thus use the terms interchangeably. For nodes n and n' , that represent plans p and p' , respectively, an index i and a time point t , if $p_i = (p', t)$, then there is an arc labeled t with index i from n to n' . Hence, the arcs in the graphic representation of plans are quadruples, (p, i, p', t) . There may be more than one arc labeled t from n to n' (in case that $p_i = p_j$, for some $i \neq j$), but they would have different indices. For simplicity, each plan p is used as a synonym for the node which is the root of its graph, and $G(p)$ is the graph containing all the descendents of p .

3.1 Properties of Structured Plans

A *plan-path* is a sequence of directed arcs in the graphic representation of plans. A *plan-path-in* p , for a structured plan p , is a plan-path that is included in $G(p)$. The *delay* of a plan-path is the sum of all delays in the path. The plan at the start of the first arc of a plan-path is its *head*, and the plan at the end of the last arc in a plan-path is its *tail*. The tail of a plan-path is always a *component* of its head. If a plan-path has a single arc, then its tail is a *direct component* of its head. For a plan-path pp with head p , tail q , and delay t , (q, t) is an *occurrence* in p via pp . If pp has a single arc, then (q, t) is a *direct occurrence*. If q is an elementary plan, then q is an *elementary component* of p , and (q, t) is an *elementary occurrence* in p via pp . A plan is *empty* if $G(p)$ has no arcs, and is *finite* if $G(p)$ is finite. In this paper we deal only with finite plans.

Example 4 *In the structured plan presented in Figure 3, $pp = \langle a_3, a_9, a_{17} \rangle$ is a plan-path in $bakeCheeseAppleCakes$, whose head is $bakeCheeseCake$, its tail is $pickFlour$, and its delay is 5. $pickFlour$ is an elementary component of $bakeCheeseCake$, and a direct component of $takeOutFlour$. $(pickFlour, 2)$ is an elementary direct occurrence in $takeOutFlour$ via $\langle a_{17} \rangle$, and $(pickFlour, 5)$ is a direct occurrence in*

3 Plan Spaces

A plan space \mathcal{P} includes *elementary* and *structured* plans, where structured plans are constructed from elementary plans and other structured plans. The intended meaning of a structured plan is that of *grouping* atomic or compound activities along a time scale. \mathcal{P} is built on top of a set OBJ of *objects*, a set OP of *action operators*, and a set T of *time points*. The formal definition follows.

Definition 3.1 A plans space \mathcal{P} is a sextuple $(OBJ, A, OP, \langle T, + \rangle, EP, P)$ with the elements defined as follows:

- OBJ is a set of elements, called *objects*, such as `location`, `hand1`, `flour`, `robot52`, etc.
- A is a non empty set of elements called *actions*.
- OP is a non empty set of primitive action operations, such as `raise`, `move`, `takeOut`, `wait`. Each action operator has an arity, say $n(\geq 0)$, and is a partial mapping from OBJ^n into A . For example, `raise(hand1)`, `move(box1, l1, l2)`, `takeOut(flour, closet)`, `wait` are actions.
- $\langle T, + \rangle$ is an additive group of time points, as described in Definition 2.1.
- EP is a set of elementary planes. Each elementary plan $ep \in EP$ is a combination of (a, d) , of an action $a \in A$, and a duration $d \in T$. For an elementary plan ep , its action is denoted **action**(ep), and its duration is denoted **duration**(ep).
- P is a domain of events, defined inductively as follows:

1. $EP \subset P$.

2. **Structured plans:**

For all $p^1, p^2, \dots, p^n \in P$, ($n > 0$), $t_1, t_2, \dots, t_n \in T$, ($n > 0$),

$plan = ((p^1, t_1), (p^2, t_1), \dots, (p^n, t_1)) \in P$, ($n > 0$).

3. Elements of P are just those obtained from 1 by a finite number of applications of 2.

For each pair (p^i, t_i) in *plan* (denoted also as $(p^i, t_i) \in plan$), p^i is a sub-plan of *plan*, and t_i is its *delay*. $plan_i$ stands for the i -th pair, and is called the *i -th element of p* . The index of a pair is its “slot” in *plan* (instead of integers we could have used a mapping from slot names to event-delay pairs). If some pairs share

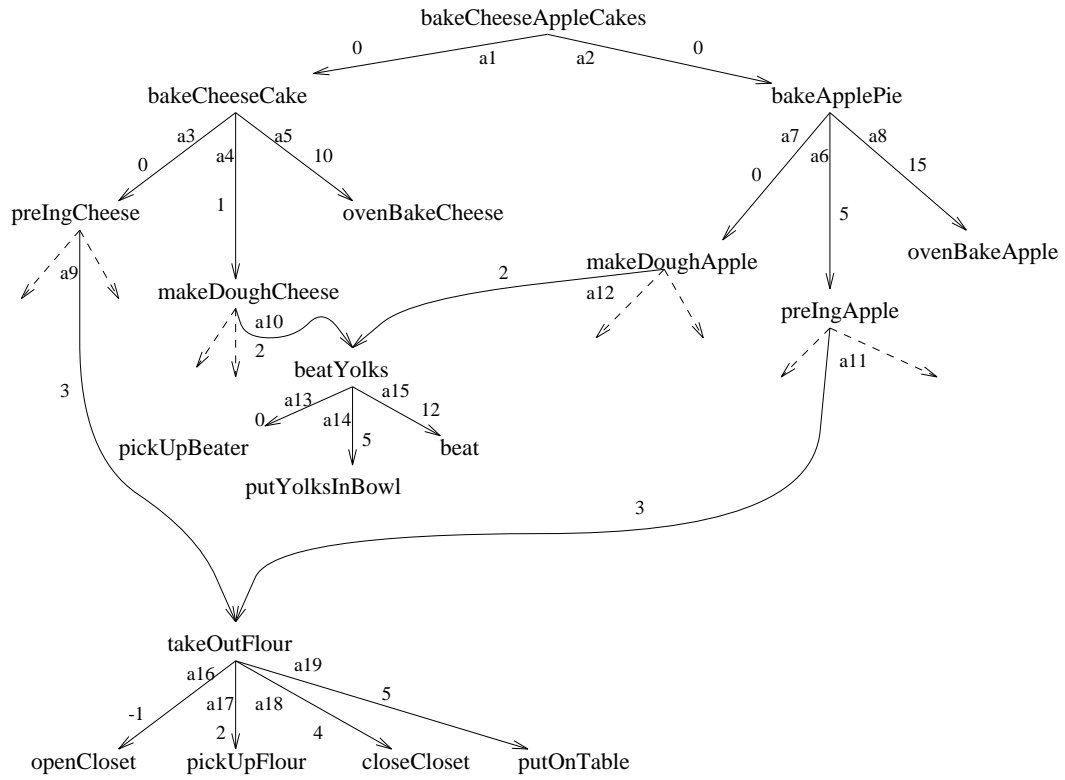


Figure 3: A structured plan

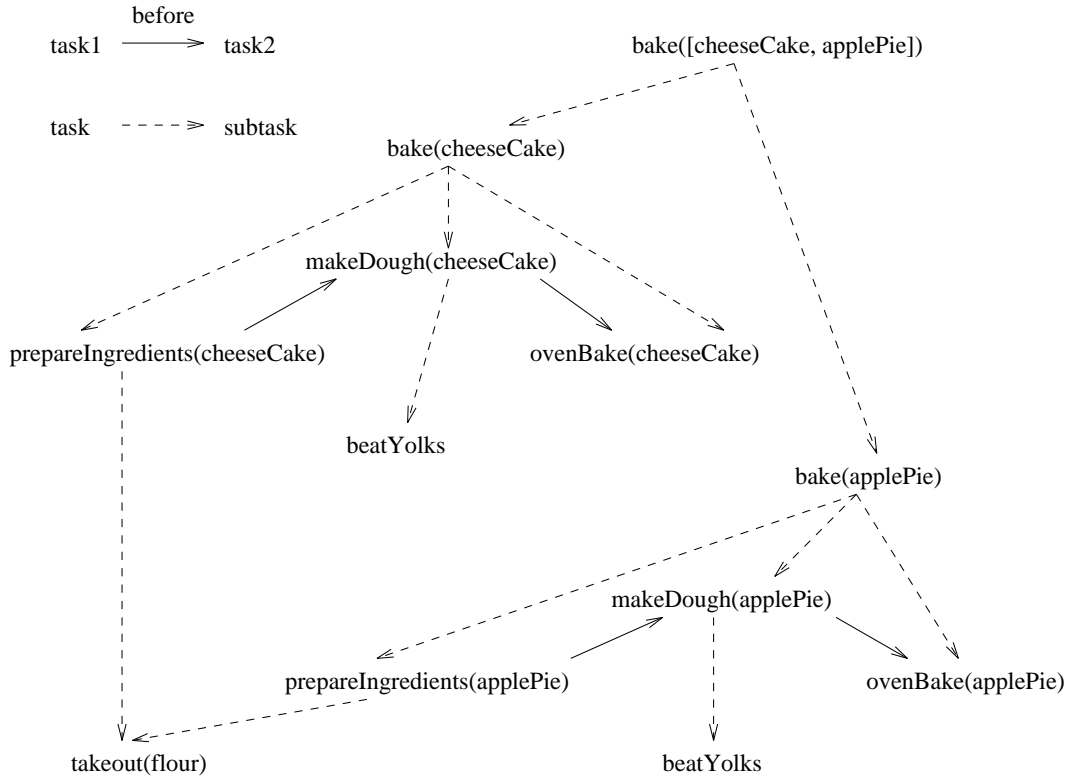


Figure 1: A task network

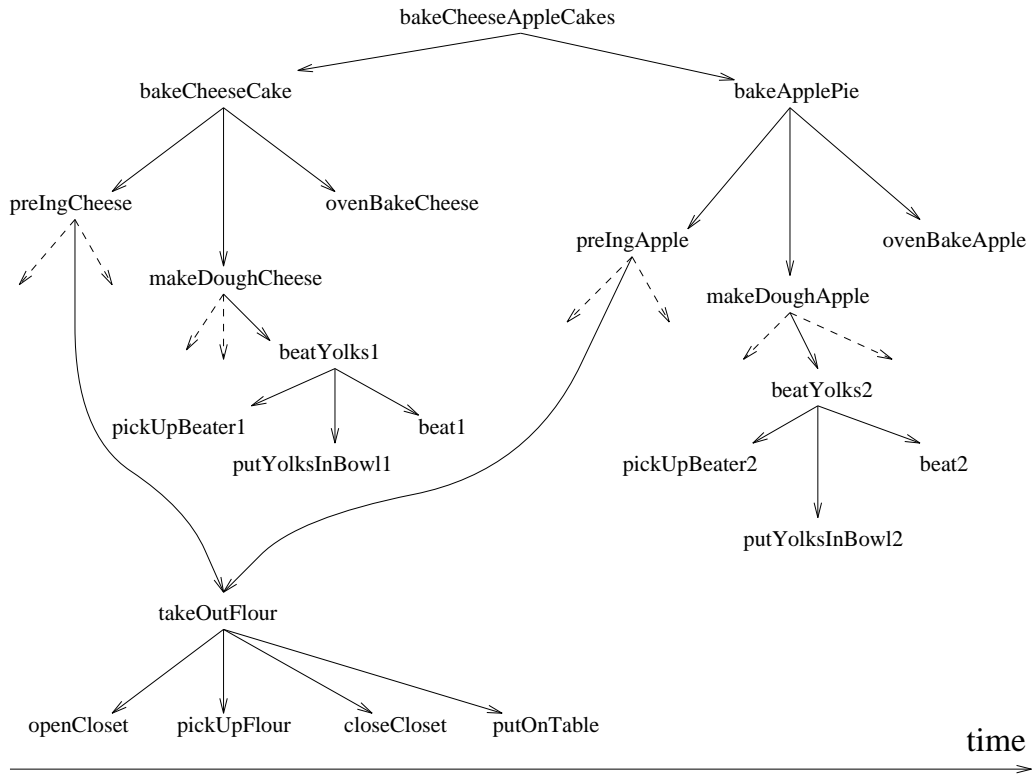


Figure 2: A concrete structured event

1. If $e_1, e_2 \in EE$ then $e_1 \neq e_2$.
2. If $e_1 \in EE$ and $e_2 = \mathbf{Se}(\nu, ts, te)$, then for all $e \in \nu$, e_1 and e are disjoint.
3. If $e_2 \in EE$ and $e_1 = \mathbf{Se}(\nu, ts, te)$, then for all $e \in \nu$, e_2 and e are disjoint.
4. If $e_1 = \mathbf{Se}(\nu_1, ts_1, te_1)$ and $e_2 = \mathbf{Se}(\nu_2, ts_2, te_2)$ then for all $e'_1 \in \nu_1$ and for all $e'_2 \in \nu_2$, e'_1 and e'_2 are disjoint.

Property 2.1

1. Disjointness relation on E is anti-reflexive.
2. Disjointness relation on E is symmetric.

Definition 2.3 A structured event $se = \mathbf{Se}(\nu, ts, te)$, where $\nu = (e_1, e_2, \dots, e_n)$, is non-sharing iff all pairs (e_i, e_j) are disjoint, and all e_i are non-sharing, for all $1 \leq i < j \leq n$. An elementary event is always non-sharing. A structured event is sharing if it is not non-sharing.

Example 3 The structured event *bakeCheeseAppleCakes* in Figure 2 is sharing, while *bakeCheeseCake* and *bakeApplePie* are non-sharing.

□

Notation: The set of all non-sharing structured events is denoted $SE/nonSharing$.

The events in E can be represented as directed acyclic graphs, where internal nodes represent structured events, leaves represent elementary plans, and there is a directed arc labeled $i (> 0)$ from node n that represents a structured event e , to node n' that represents an event e' , iff $se = \mathbf{Se}(\nu, ts, te)$, and $e = \nu_i$. For simplicity, each event e is used as a synonym for the node which is the root of its graph, and $G(e)$ is the graph containing all the descendents of e (i.e., all nodes reachable from e via a directed path). The arcs in the graphical representation of E are the triplets $\{(e, i, e') \mid e = \mathbf{Se}(\nu, ts, te), e' = \nu_i\}$.

2.1 Properties of Structured Events

The *direct component* and *component* relations between structured events reflect their structure: For $se = \mathbf{Se}(\nu, ts, te)$, each $e \in \nu$ is a *direct component* of se ; the *component* relation is the transitive closure of the *direct component* relation. An *event-path* is a sequence of directed arcs in the graphic representation of events. An *event-path-in* e , for a structured event e , is an event-path that is included in $G(e)$. Hence, there is an arc from an event to each of its direct components, and there is an event-path from an event to each of its components. An *elementary component* of a structured event e is a component which is an elementary event. A structured event is *flat* if all of its direct components are elementary. An elementary event is considered as *flat* as well. A structured event e defines a single *flat event* e_{flat} , whose direct components are all elementary components of e , in the same order. Graphically speaking, e_{flat} has all leaves of e as direct components.

Example 2 *In the structured event described in Figure 2, takeOutFlour is a direct component of preIngCheese, and is a component of bakeCheeseCake; beat1 and beat2 are elementary components of bakeCheeseAppleCakes; the sequence of arcs through the nodes bakeCheeseCake, preIngCheese, takeOutFlour is a length two event-path in bakeCheeseAppleCakes.*

□

Disjointness and sharing:

Definition 2.2 *Events e_1 and e_2 are disjoint iff:*

property removed.
similarity property
removed.

Definition 2.1 An event space \mathcal{E} is a quintuple $(A, \langle T, + \rangle, EE, E)$ with the elements defined as follows: \oplus was omitted

- A is a set of elements called actions.
- $\langle T, + \rangle$ is an additive group, where $+$ is a binary operation on T . 0 is the zero element of $\langle T, + \rangle$, and $(-t)$ is the inverse of t . The set T is totally ordered; elements preceding 0 are negative; elements succeeding 0 are positive. We also single out elements called durations, which denote elapsed time. They behave like non-negative time points. Henceforth we identify durations with non-negative time points.
- EE is a domain of elementary events. Each elementary event $ee \in EE$ has an associated action $\mathbf{action}(ee)$, a start time point $\mathbf{start}(ee)$, and an end time point $\mathbf{end}(ee)$. The duration of ee , denoted $\mathbf{duration}(ee)$, is $\mathbf{end}(ee) - \mathbf{start}(ee)$.
- E is a domain of events, defined inductively as follows:

1. $EE \subset E$.

2. **Structured events:**

Let \mathcal{S} be the set of all tuples (e_1, e_2, \dots, e_n) ($n > 0$, $e_i \in E$), and let $\mathcal{S}' \subseteq \mathcal{S}$, an arbitrarily changed. \mathcal{S}' is a selected subset. Then, for all $\nu = (e_1, e_2, \dots, e_n) \in \mathcal{S}'$, if $ts = \min\{\mathbf{start}(e_i)\}_{i=1}^n$ and $te = \max\{\mathbf{end}(e_i)\}_{i=1}^n$, then ts is the earliest time, and te is the latest.

$$\mathbf{Se}(\nu, ts, te) \in E.$$

The temporal functions are extended as follows: $\mathbf{start}(e) = ts$, $\mathbf{end}(e) = te$, $\mathbf{duration}(e) = te - ts$.

3. Elements of E are just those obtained from 1 by a finite number of applications of 2.

If in every application of step 2, we have $\mathcal{S}' = \mathcal{S}$, then E is complete over the set of elementary events EE . Next item is o

Notation: $SE = E - EE$, is the set of *structured events*. Henceforce, the term *structured event* refers only to elements of SE , while the term *event* refers to all, elementary and structured events. removed sente

For an n-tuple ν , $|\nu|$ denotes the size of the tuple n , ν_i is the i-th element of ν , and $e \in \nu$ states that e is an element of ν .

Suppose that we wish to add a plan for obtaining a cheese cake and an apple pie, saying that we can do that either by baking, or by buying the cakes in a near by conditory. We add the following schema:

$$\text{obtainCheeseAppleCakes} := (\text{bakeCheeseAppleCakes} ; \text{buy}([\text{cheeseCake}, \text{applePie}], \text{nearByConditory}))$$

Here “buy” is the name of another plan schema that must be defined. It takes two arguments. The symbols “cheeseCake”, “applePie”, and “nearByConditory” are object symbols. They can also be defined, as in:

$$\text{nearByConditory} \stackrel{\text{def}}{=} [\text{Kapulsky}, \text{Roval}, \text{Pitput}]$$

The similarity to Prolog notation is no coincidence; indeed “,” stands for conjunction and “;” stands for disjunction.

□

The Plan Structures approach can be used by planners in the standard way: Starting with a goal plan schema, a task network can be incrementally constructed, based on the plan definition, and other standard information like rules. The task network partially describes structured plans denoted by the goal plan schema. The contribution of the Plan Structures approach is in introducing a formal framework of plan schemas, a framework that allows us to specify sharing and non-sharing constraints, and can provide a yard stick for correctness of planning algorithms.¹

Section 2 introduces event spaces, and in Section 3 plan spaces and their meanings are defined. Section 4 introduces the Plan Structures Language for specification of plan schemas. Section 5 discusses issues related to the usage and management of a database of plan schemas, written in the Plan Structures Language. Section 6 is the Conclusion. Proofs appear in the Appendix.

2 Event Spaces

An event space \mathcal{E} includes *elementary* and *structured* events, where structured events are constructed from other elementary and structured events. \mathcal{E} is built on top of a set A of *actions*, and a set T of *time points*. The actions act as the *types* of the elementary events, and the time points specify the temporal properties of events. The formal definition follows.

¹A view of complex activities as structured objects was introduced in [3, 4]. Complex activities over time were captured by a data model of structured histories; complete specifications were enabled by the Time Structures language.

plan. The shifts are written on the arcs; arcs have identities (denoted a_i). A structured plan denotes all structured events that agree with its structure and temporal relationships among components.

A structured plan is associated with a *Sharing-Constraint-System (SCS)*. The *SCS* specifies component structured plans that, necessarily, denote shared structured events, or that, necessarily, denote distinguished structured events. For example, in Figure 3, in every structured event denoted by the *bakeCheeseAppleCakes* structured plan, there should be a single component structured event that is denoted by the *takeOutFlour* structured plan, i.e., it is *shared* by the two *prepareIng* structured plans. Hence, the *SCS* of *bakeCheeseAppleCakes* includes the *positive Sharing-Constraint (SC)* $\{\langle a_1, a_3, a_9 \rangle, \langle a_2, a_6, a_{11} \rangle\}$. Also, in every structured event denoted by *bakeCheeseAppleCakes*, there should be two distinguished component structured events denoted by the *beatYolks* structured plan, i.e., the two *makeDough* structured plans, necessarily, do not share their *beatYolks* component. Hence, the *SCS* of *bakeCheeseAppleCakes* includes also the *negative SC* $\{\langle a_1, a_4, a_{10} \rangle, \langle a_2, a_7, a_{12} \rangle\}$.

Plan schemas are symbolic expressions that denote structured plans, and their associated *SCSs*. Plan schemas have the following parts:

1. Plan structure definitions: Denote structured plans.
2. *CSs* expressions: Denote collections of positive and negative *SCs*.
3. Standard parts of plan schemas, such as *protections* and *preconditions*. We do not deal with these parts in this paper, and omit them from our discussions of plan schemas.

Example 1 presents several plan schemas, for the baking structured plan.

Example 1

$$\begin{aligned}
 \textit{bakeCheeseAppleCakes} &:= \mathbf{and}(\textit{bakeCheeseCake}, \textit{bakeAppleCake}) \\
 &\quad \textit{positive SC} : \textit{bakeCheeseAppleCakes}, * \textit{takeOutFlour} \\
 &\quad \textit{negative SC} : \neg \textit{bakeCheeseAppleCakes}, * \textit{beatYolks} \\
 \textit{bakeCheeseCake} &:= \mathbf{before}(\textit{preIngCheese}, \textit{makeDoughCheese}, \textit{ovenBakeCheese}) \\
 \textit{bakeApplePie} &:= \mathbf{before}(\textit{preIngApple}, \textit{makeDoughApple}, \textit{ovenBakeApple}) \\
 \textit{preIngCheese} &:= \mathbf{partial_and}(\textit{takeOutFlour}, \textit{takeOutCheese}, \textit{takeOutSugar}) \\
 \textit{makeDoughCheese} &:= \mathbf{before}(\textit{separateYolks}, \textit{beatYolks}), \\
 &\quad \mathbf{partial_and}(\textit{separateYolks}, \textit{beatYolks}, \textit{mixFlours})
 \end{aligned}$$

three levels, as shown in the following figure:

PLAN SCHEMAS : Expressions of a Plan Structures Language

↓ *denote*

STRUCTURED PLANS

↓ *denote*

CONCRETE STRUCTURED EVENTS

Concrete structured events are real world occurrences, *structured plans* are abstractions that extract the common structure of occurrences, and *plan schemas* are symbolic expressions that describe, in part or in full, structured plans. The meaning of plan schemas in the real world is given indirectly, via structured plans.

Concrete structured events are constructs that organize events into natural structures. For example, referring to the example in Figure 1, a concrete baking of a cheese cake and an apple pie is a structured event. Part of such an event is graphically described in Figure 2.

Insert Figure 2

Each concrete structured event has a time span which is a time interval in which it occurred. Structured events like *takeOutFlour* that are shared by several structured events are *components* of these events. The common structure of different structured events, such as *beatYolks1* and *beatYolks2*, is not singled out.

Structured plans are stand alone constructs that abstract away from the concrete occurrences of structured events, and account only for their structure. Part of a structured plan that can denote the baking structured event from Figure 2 is graphically described in Figure 3.

Insert Figure 3

Note that the two structured events *beatYolks1* *beatYolks2* are now captured by the single structured plan *beatYolks*. The independence of structured plans is achieved by associating each structured plan with an independent clock. The timing relationships between a concrete occurrence and its components are extracted by shifting the clocks of the component structured plans with respect to the clock of the parent structured

1 Introduction

Hierarchical planning has been with us for a while. Beginning with NOAH [?] and to some extent ABSTRIPS [?], enhancements occurred later on to introduce backtracking to partial plan generation in NONLIN [?], and then SIPE [?], and many others. In hierarchical planning, one starts with a goal to be achieved, decomposes it into subtasks to be achieved, ending in primitive actions to be executed. Different planners perform this in different ways, but all of them use an abstract plan library of some sort, and search for ways to decompose high level plans by using the library. Since there is more than one way to perform each decomposition, or to reduce a goal, there is a search in the space of partial (hierarchical) plans. Planners keep track of the tasks and subtasks by using a *task network*, which is a graphical structure showing task decomposition and temporal ordering constraints. As far as time is concerned, a task network is a hierarchical *time map*. Figure 1 presents an example task network. It shows that the task of baking a cheese cake and an apple pie can be decomposed into two tasks for baking each cake. The baking of each cake can be further decomposed into three successive steps involving the preparation of ingredients, making the appropriate dough, and baking in the oven. Making the dough, in each cake, includes a task of beating egg yolks, and the two tasks of ingredients' preparations share a common task for taking the flour out from the closet.

Insert Figure 1

In a task network, if a task *tsk* is a subtask of more than one parent task (in cases where this is allowed by the planner at all), then it is interpreted as a single occurrence of *tsk*, i.e. that subtask is shared by both parents and will occur exactly once during execution. If a task type can occur in numerous places in the task network, it may be advantageous to allow a subtask *tsk* to represent *more than one* occurrence of the task during execution, for conciseness. However, if the latter is allowed (as it is in structured plans), it should be made clear how many distinct actual occurrences of *tsk* exist, i.e. to what extent is *tsk* *shared* among its parent tasks.

In this paper we introduce a representation framework for plans, that accounts for *temporal* and *hierarchical* aspects, and supports *repetition* and *sharing* of sub-plans. The framework, called *Plan Structures*, is in

Structured Plans with Sharing and Repetition: Model and Specification

Mira Balaban Solomon Eyal Shimony

Math. and Computer Science Dept.

Ben-Gurion University

P.O. Box 653

84105 BEER-SHEVA, ISRAEL

Abstract