Taxonomic plan reasoning
Premkumar T. Devanbu*, Diane J. Litman
Artificial Intelligence Principles Research Department, AT&T Bell Laboratories, 600 Mountain Avenue,
Murray Hill, NJ 07974, USA
Received October 1991; revised June 1995

Abstract
CLASP (CLASSification of Scenarios and Plans) is a knowledge representation system that extends the notion of subsumption from terminological languages to plans. The CLASP representation language provides description-forming operators that specify temporal and conditional relationships between actions represented in CLASSIC (a current subsumption-based knowledge representation language).

CLASP supports subsumption inferences between plan concepts and other plan concepts, as well as between plan concepts and plan instances. These inferences support the automatic creation of a plan taxonomy. Subsumption in CLASP builds on term subsumption in CLASSIC and illustrates how term subsumption can be exploited to serve special needs. In particular, the CLASP algorithms for plan subsumption integrate work in automata theory with work in term subsumption.

Keywords: Knowledge representation; Artificial intelligence; Reasoning; Reuse

1. Introduction
Terminological systems [30] are in the KL-ONE family [9] of knowledge representation languages, and provide representational support in many areas of artificial intelligence.† Central to terminological approaches are the use of classification and term subsumption inferences‡ to organize frame taxonomies, and differentiation between terminological and assertional aspects of knowledge. A major limitation of current ter-

* Corresponding author. E-mail: prem@research.att.com.
† Terminological systems have most recently been called description logics in the literature.
‡ We assume the standard interpretation for these terms, as described in [8]. Briefly, subsumption determines whether one frame is more general than other. Classification uses subsumption to find the correct place for a frame in a taxonomy.
minological systems, however, is an inability to represent and reason with plans. Plans, compositions of actions that achieve given goals, play a central role in many areas that use terminological knowledge representation systems (natural language processing [34], expert systems [35], user interfaces [14, 38], plan synthesis [42], software information systems [11]). While the generation and recognition of plans has been the focus of much research in automatic reasoning, the knowledge representation task of managing collections of plans has largely been unaddressed. In this paper we present a knowledge representation and reasoning system that extends the notion of subsumption from terminological languages to plans. The many well-known benefits that are obtained by representing knowledge in standard terminological systems [9, 25] can thus be obtained in the domain of plan representation as well.

Our development of a plan-based terminological representation system was motivated by the application of terminological systems to software information systems, and the need to represent plans in the domains modeled by such systems. Fischer and Schneider [16] suggested that a large software project should use a knowledge base to collect and disseminate information about all aspects of the system under construction, e.g., the specific domain of application, the architecture of the system, or how the architecture is designed to service the needs of the application domain. The LASSIE software information system [11] pursued this idea, using the terminological knowledge representation language CLASSIC [8] to describe the architecture, domain model, and code of the AT&T Definity™ 75/85, which is a scalable private branch exchange (PBX) switching product. By using a terminological language, LASSIE could use classification to organize descriptions into a taxonomy and to do retrieval based on the semantics of the query and the stored descriptions. A taxonomy of the actions and objects in the telephony domain formed the core of the LASSIE knowledge base. However, LASSIE had representational needs beyond the capabilities of terminological systems. Because terminological languages (including CLASSIC) have no meaningful way to represent or reason with plans (temporal compositions of actions), LASSIE could not be used to describe sequences of actions that achieved particular goals (e.g., call forwarding, as found in most modern telephone systems). Context-dependent, temporal, and other relationships could not be captured. In addition, action subsumption and classification did not support standard inferences found in planning systems. For example, determining how an action changes the state of the world depends on reasoning specific to action roles such as preconditions and effects.

Motivated by such issues, we have designed and implemented a plan-based terminological knowledge representation system called CLASP (CLAssification of Scenarios and Plans). CLASP is designed to represent and reason with large collections of plan descriptions, much in the same way current terminological systems reason with object descriptions. CLASP creates plan descriptions from action and state descriptions, using a restricted plan language containing temporal and conditional operators. In particular, CLASP plans are composed from actions and states using an extension of regular expressions. CLASP uses the semantics of these descriptions to associate plan descriptions with sets of plan individuals, and to organize plans into taxonomies based

---

1 LASSIE was originally implemented using the terminological language KANDOR [28].
solely on terminological inferences. The terminological inferences are computed by extending standard algorithms involving finite automata, such that transitions correspond to subsumption rather than equality checks. CLASP thus extends the terminological paradigm to plans, by integrating work in automata theory with work in term subsumption.

Fig. 1 shows an example of a simple CLASP plan taxonomy. Plan-Class1 and Plan-Class2 are plan descriptions. The internal arrows show temporal and conditional relationships between action descriptions (e.g., Goto) that compose the plan descriptions. Plan individuals or instances are specific sequences of action instances. Plan classes correspond to sets of plan instances that satisfy the descriptions. For example, Plan-Class1 describes all plan instances in which an instance of a Goto action temporally precedes an instance of an Attend-Conference action, which temporally precedes zero or more sequential instances of the action Visit-Research-Institution, which temporally precede an instance of the action Leave. plan-instance represents a temporal sequence of the three action instances shown. The CLASP terminological inferences use the semantics of the representations to organize plan classes and instances into a taxonomy. For example, plan-instance is an instance of Plan-Class1 because (1) the action instances 5/22-fly-to-Germany, attend-KR94, and 5/28-fly-from-Germany are (abstractly) described by the action classes Goto, Attend-Conference, and Leave, and (2) the sequence of the three actions in plan-instance satisfies the temporal and conditional constraints of Plan-Class1. In contrast plan-instance is not an instance of Plan-Class2, because there is no action individual described by the non-optional action class Visit-University. Finally, plan subsumption can compute that Plan-Class2 is a subclass of Plan-Class1, because any plan instances described by Plan-Class2 are necessarily also described by Plan-Class1.

CLASP is designed as a companion to the CLASSIC [8] terminological knowledge representation system, in order to allow CLASP to make use of CLASSIC'S well-defined sub-
sumption inference. Thus, in the above example, CLASSIC term subsumption can be used to infer that 5/22-fly-to-Germany is an instance of Fly-In, 5/22-fly-to-Germany is an instance of Goto, Fly-In is a subclass of Goto, and so on. To determine the instance and subclass relationships between the plans, the CLASP algorithms then use results in automata theory to extend the terminological inferences to plans. Indeed, one contribution of our work is to demonstrate how term subsumption can serve as a springboard for implementing special purpose representation and reasoning mechanisms.

In the next section we further motivate CLASP by detailing the importance of taxonomic reasoning with plan-like knowledge. In Section 3 we describe the details of the CLASP system, an integration of automata and terminological theory. We first discuss the representation language for representing plan descriptions and plan instances, then discuss the terminological inference mechanisms for computing the plan subsumption and instance inferences. In Section 4 we show an application of CLASP in extending what can be represented in the domain model of a terminological software information system. In Section 5 we discuss limitations of CLASP with respect to representing and reasoning with more expressive plan languages, and discuss how CLASP could be extended to handle a particular type of iteration construct. Finally, in Sections 6 and 7, we relate CLASP to existing research in taxonomic plan reasoning, and conclude with some thoughts on future directions.

2. Why terminological plan representation and reasoning?

The advantages obtained from representing taxonomic knowledge in terminological systems are well documented [9,25]. Terminological systems endow a taxonomy with a formal semantics. Classification via subsumption lets the system rather than the user organize the taxonomy and has many benefits from a knowledge engineering perspective. For example, a terminological system can automatically detect incoherent or duplicate definitions, and can perform retrieval based on the semantics of a query. Unfortunately, these advantages can only be realized in limited domains and applications, as the current expressive power of most terminological systems is fairly limited [13].

Plans play a central role in many domains that are already partially represented using terminological systems. Extending the scope of the terminological paradigm to plans would provide an integrated framework for terminological and plan-based representation and reasoning, and bring the advantages of the terminological approach to the area of plan representation. For example, in CLASP, subsumption in an existing terminological system provides the semantics for the building blocks of plans, while plan subsumption formalizes and automates the taxonomic organization of the plan knowledge base. From a more practical perspective, CLASP supports the construction of a plan knowledge base using components taken from an existing terminological ontology. In addition, plan subsumption supports the development of a terminological plan-based information system, in the style of existing terminological information systems [11,28], which support semantic-based retrieval.
The advantages of endowing a taxonomy with formal semantics could also be extended to existing non-terminological plan representations that are already taxonomic. For example, although plan taxonomies are used in systems that perform both plan recognition [21,40] and plan synthesis [36,42], to date only Weida and Litman [40] and Wellman [42] use terminological reasoning to provide semantics to the plan taxonomies and the building blocks of the plans.

Finally, the plan subsumption inference that is inherent in a terminological plan representation system can provide the foundation for fundamentally new approaches to other tasks in automatic reasoning. Promising work along these lines has already been demonstrated in the areas of plan synthesis [42] and plan recognition [40,41].

As discussed above, the need to represent and retrieve plan-like knowledge in the terminological framework of the LASSIE software information system was the original impetus for the development of CLASP. LASSIE needed to represent plan-like knowledge in the domain of telephone switching software, to extend the use of LASSIE in software development tasks such as specification, testing and debugging. For example, plan-like structures are particularly useful in specifying features such as "call forwarding" and "call waiting". While a full description of a feature describes behavior under differing conditions, a feature is often illustrated in terms of a feature scenario representing just one aspect of the behavior. Thus, a scenario illustrating one successful use of "call waiting" might be: "A picks up the phone, gets dial-tone and dials B; since B is off-hook, A gets a special ringing tone and B gets a call-waiting signal; B flashes hook and connects with A". A full feature description is a generic description of actions and associated goals, much like a plan, while a scenario is a specific manifestation much like a plan execution trace. In a modern telephone switch, there are a large number of features; subsumption with feature descriptions is helpful, as we describe below.

Plan-like structures are also used during the testing phase of software development. In switching software, tests are usually represented as test scripts. Test scripts specify stimuli to the switch along with expected responses, for example, "Pick up the phone; the system produces a dial-tone". A test script is thus representationally very much like a scenario description described above, i.e., a plan-like series of actions. Given the number of test scripts—a large project can have on the order of 10,000 scripts—the ability to represent and manage such scripts within a LASSIE-like system would be useful.

Plans can also be useful in explaining the behavior of distributed software systems, like the AT&T Definity™ 75/85 switch. The processing of a typical stimulus to a switch involves the exchange of messages between several processes. These messages are logged to a file and examined off-line. The comprehension of these message traces is an important step in understanding the software, and involves constructing explanations that are very plan-like. For example, a message trace might show a request to a trunk handling process to open a connection to another switch; this might be followed by a series of messages requesting packet transmission, followed by another message to close the transmission. The explanation of this trace would be a plan with a sequence of actions: the first action connects the trunk and makes it ready for transmission; a series of actions send the messages; a final action closes the connection. It would be useful to create and store such explanations for subsequent use.
3. CLASP

CLASP is a plan-based knowledge representation system that extends and builds upon CLASSIC, the term subsumption system used in LASSIE. That is, just as CLASSIC allows users to define descriptions and create instances of terms, CLASP allows users to define plan concepts and create scenario instances. Similarly, just as subsumption and classification are the central inferences in CLASSIC, plan subsumption and classification are the core inferences in CLASP. In particular, CLASP can compute the generalization relationships that organize plan concepts into taxonomies, and can associate plan concepts with sets of scenarios (plan individuals). To integrate the two representation systems, we use the conceptual framework shown in Fig. 2. All plan operations are handled by CLASP, which itself internally calls CLASSIC.

Section 3.1 introduces the use of CLASSIC in representing terms. Section 3.2 presents the CLASP representation language that allows plans and scenarios (the nodes of the CLASP taxonomy) to be compositionally defined from CLASSIC terms. Section 3.3 presents the algorithms computing terminological inferences involving plans and scenarios (computing the arcs in the CLASP taxonomy), as well as algorithms for additional types of plan-based reasoning. Complexity analyses associated with our algorithms are also presented. As we will see, the plan-based inferences of CLASP use results in automata theory to extend the capabilities already available in CLASSIC.

3.1. Terms: the building blocks of CLASP

CLASP complements the term language of CLASSIC by providing plan operators to form plans from CLASSIC terms. In particular, since actions and states are the main building blocks of CLASP plans and scenarios, the representation of actions and states in
CLASSIC is briefly discussed here. As we will see, by defining plans in terms of CLASSIC concepts, the algorithms for plan classification can take advantage of CLASSIC's terminological inferences.

Frames in CLASSIC are called concepts. They are (potentially complex) descriptions and are formed by restricting other descriptions using a small set of description-forming operators. For example, existing concepts can be conjoined using the operator "AND". Roles represent properties and can also be further constrained. The "ALL" value restriction restricts all fillers of a particular role to be of a certain type, while the number restrictions "AT-LEAST" and "AT-MOST" specify constraints on the number of fillers for a particular role. "FILLS" specifies particular individuals that fill the role. While there are other types of restrictions in CLASSIC, they will not be needed to understand the examples in this paper. Individuals are specific instances of concepts, and are created using the same restrictions as for concepts. The extension of a concept is the set of individuals described by the concept.

CLASP uses CLASSIC to define built-in concepts Action, State, and Agent. For example, CLASP defines the CLASSIC concept Action to represent an action operator in the style of STRIPS [15], where role restrictions specify the characteristics of the roles ACTOR, PRECONDITION, ADD-LIST, DELETE-LIST, and GOAL. In particular, an action is defined thus:

\[
(\text{DEFINE-CONCEPT} \neg Action \\
(\text{PRIMITIVE} \neg (\text{AND} \neg \text{Classic-Thing} \\
(\text{AT-LEAST} \ 1 \ \text{ACTOR}) \\
(\text{ALL} \ \text{ACTOR} \ \text{Agent}) \\
(\text{EXACTLY} \ 1 \ \text{PRECONDITION}) \\
(\text{ALL} \ \text{PRECONDITION} \ \text{State}) \\
(\text{EXACTLY} \ 1 \ \text{ADD-LIST}) \\
(\text{ALL} \ \text{ADD-LIST} \ \text{State}) \\
(\text{EXACTLY} \ 1 \ \text{DELETE-LIST}) \\
(\text{ALL} \ \text{DELETE-LIST} \ \text{State}) \\
(\text{EXACTLY} \ 1 \ \text{GOAL}) \\
(\text{ALL} \ \text{GOAL} \ \text{State}))))
\]

The above definition states that "An Action is a Classic-Thing, with at least one ACTOR, all of whose ACTORS are of type Agent, whose PRECONDITION is of type State, whose ADD-LIST is of type State, whose DELETE-LIST is of type State, and whose GOAL is of type State" (assuming the previous definition of the concepts Classic-Thing, State and Agent in CLASSIC, as detailed below). EXACTLY (followed by a number) is our notation for an operator that defines precisely how many fillers are allowed for a slot. In CLASSIC this would be expressed using AT-MOST

---

4 In our informal notation, Concept-Names will be shown in capitalized typewriter font, individual-names in lower-case typewriter font, ROLE-NAMES in upper-case typewriter font, and CLASSIC-OPERATORS (and later CLASP-OPERATORS) in upper-case.
and AT-LEAST operators with the same number.\(^5\) This example has the role restrictions that there must be at least one filler of the role ACTOR, that all fillers of the role ACTOR must be of type Agent, and so on. The PRIMITIVE operator is used to specify that the concept definition does not fully specify sufficient conditions for class membership, so classification of other concepts underneath a primitive must be explicitly licensed.

The concept State is also pre-defined by CLASP—as a primitive CLASSIC concept specializing Classic-Thing:

\[
(\text{DEFINE-CONCEPT}) \\
\text{State} \\
(\text{PRIMITIVE} \text{Classic-Thing})
\]

Furthermore, CLASP allows States to only be restricted using the CLASSIC description-forming operators PRIMITIVE and AND. In other words, CLASP restricts the user to only the following subset of the CLASSIC syntax when defining States:

\[
\text{(state-concept)} ::= \\
(\text{PRIMITIVE} \text{(state-concept)}) \mid \\
(\text{AND} \text{(state-concept)} +) \mid \\
\text{State}
\]

All state descriptions can thus be reduced to a simple conjunction of primitive descriptions of type State, using the CLASSIC normalization procedure of eliminating embedded AND operators. Our representation will facilitate STRIPS-like tracking of state information, as discussed in Section 3.3.3.

Note that our terminological representation of states is not as expressive as the predicate calculus representation used in STRIPS. However, by using a terminological representation of states, we can avoid doing general theorem proving when computing state subsumption. In particular, since states are just a subset of CLASSIC concepts, state subsumption is computed using the subsumption algorithms of CLASSIC. Terminological systems in general highly restrict the expressive power of their representation languages (typically to a subset of predicate calculus), in order to improve the computational complexity and/or the completeness of performing inferences such as subsumption. For example, CLASSIC does not contain such predicate calculus operators as disjunction, negation, and existential quantification.\(^6\) See \cite{7,26} for more general discussions of the relationship between predicate calculus and terminological systems. Within the family of terminological systems, CLASSIC is in fact one of the most restricted systems with respect to expressive power. If CLASP was built on a different terminological system, the expressive power of the state description language would still

---

\(^5\) As we will see, multiple PRECONDITIONs and GOALS are specified using State concepts that are conjunctions of other state concepts. Similarly, although we use the terms ADD-LIST and DELETE-LIST for historical reasons, the “lists” are actually conjunctions of state concepts.

\(^6\) There are, however, “tricks of the trade” for representing limited forms of negation, disjunction, and other constructs in CLASSIC \cite{8}.
be restricted compared to predicate calculus, but in different and less severe ways. For example, CLASSIC has extremely limited facilities for expressing equality, which cannot be used to represent the necessary constraints among the roles of states. We have thus made the state representation in CLASP propositional (i.e., states do not have roles). In other words, due to the inability to express complex role relationships in CLASSIC, in our domain we encode information about roles using concepts rather than role restrictions. While this is adequate for our representational needs, if we were to build CLASP on top of a more expressive terminological system, we could alternatively use a state description language such as proposed in [19]; however, this could complicate the subsumption algorithms. Propositional versions of STRIPS can also be found in the planning literature [5], also motivated by issues of computational complexity.

Actions and States can be restricted to define various specializations in the LASSIE domain. As we will see, these action and state concepts can be combined in CLASP to form plans, while the individual instances of the actions can be combined to form scenarios. For example, the Action concept specializes into System-Acts and User-Acts. System-Act is defined below:

\[
(\text{DEFINE CONCEPT} \quad \text{System-Act} \quad (\text{AND Action} \quad (\text{ALL ACTOR System-Agent})))
\]

This definition declares that System-Act is a subconcept of Action, where all the fillers of the (inherited) role ACTOR are restricted to be individuals described by the concept System-Agent (which itself must be defined in CLASSIC as a subconcept of Agent). Unlike the primitive concept Action, this concept is fully specified by necessary and sufficient conditions.

The definition of System-Act can itself be restricted: 8

\[
(\text{DEFINE-CONCEPT} \quad \text{Connect-Dialtone-Act} \quad (\text{AND System-Act} \quad (\text{ALL PRECONDITION} \quad (\text{AND Off-Hook-State \quad Idle-State}))) \quad (\text{ALL ADD-LIST Dialtone-State}) \quad (\text{ALL DELETE-LIST Idle-State}) \quad (\text{ALL GOAL} \quad (\text{AND Off-Hook-State \quad Dialtone-State})))
\]

---

7 While the system of [26] supports a full first order predicate calculus representation, the tradeoff is that classification is incomplete.

8 In the remainder of this paper, state descriptions will generally be shown using the normalized form of a conjunction of primitive states.
Informally, the system performs a Connect-Dialtone-Act and generates a dialtone after a user picks up a phone. Notice that this concept is defined by specifying more properties that restrict System-Act.

We can also define domain specializations of states. For example, we can describe the state of a world with two phones, where one is off-hook and the other is on-hook, by building up the following definitions:

\[(\text{DEFINE-CONCEPT})\]
\[
\text{Callee-Off-Hook-State} \\
(\text{PRIMITIVE State})
\]

\[(\text{DEFINE-CONCEPT})\]
\[
\text{Caller-On-Hook-State} \\
(\text{PRIMITIVE State})
\]

\[(\text{DEFINE-CONCEPT})\]
\[
\text{Callee-Off-Caller-On-State} \\
(\text{AND Callee-Off-Hook-State} \\
\text{Caller-On-Hook-State})
\]

For clarity, we reflect the propositional encoding of roles of states (e.g. Callee, Caller) in the state name.

CLASSIC can also be used to create individuals that are described by (i.e., are specific instances of) concepts. In particular, an individual must satisfy the restrictions of the describing concept. The following CLASSIC function creates an individual act1 that is described by the concept System-Act (defined above):

\[(\text{CREATE-IND})\]
\[
\text{act1} \\
(\text{AND System-Act} \\
(\text{FILLS ACTOR switching-system}))
\]

Note that for act1 to satisfy the restrictions of System-Act, the specified filler of the role ACTOR (the individual switching-system) must itself have been previously created, and must be describable by the concept System-Agent.

The following CLASSIC function creates an individual initial-state that is described by the concept Callee-Off-Caller-On-State (defined above):

\[(\text{CREATE-IND})\]
\[
\text{initial-state} \\
\text{Callee-Off-Caller-On-State}
\]

Recall that when normalized, a state description is always a conjunction of primitive, propositional state descriptions. This allows CLASP to determine if a particular proposition P holds in a state instance of type S by determining if S is subsumed by P. For example, the callee is off-hook in initial-state because CLASSIC can determine that Callee-Off-Hook-State subsumes Callee-Off-Caller-On-State (a conjunction of Callee-Off-Hook-State and Caller-On-Hook-State).
3.2. CLASP representation

3.2.1. Plans

CLASP provides a representation language for plans, descriptions that organize and group together context-independent CLASSIC action and state descriptions. Thus, a CLASSIC action type such as Connect-Dialtone-Act will occur in many CLASP plans. As we will see, by being defined in terms of CLASSIC concepts, the algorithms for plan classification can take advantage of CLASSIC inheritance and subsumption.

A plan is defined in CLASP by specifying a name and restricting the roles PLAN-EXPRESSION (a plan concept expression, specified using the syntax below), and optional roles INITIAL and GOAL. The roles INITIAL and GOAL can only be restricted by CLASSIC state concepts, as defined in the previous section. Plan concept expressions are compositionally defined from CLASSIC action and state concepts using the plan description-forming operators SEQUENCE, LOOP, REPEAT, TEST, OR, and SUBPLAN:

\[
\langle \text{plan-concept-expression} \rangle ::= \\
(\text{action-concept}) \mid \\
(\text{SEQUENCE} \langle \text{plan-concept-expression} \rangle^+) \mid \\
(\text{LOOP} \langle \text{plan-concept-expression} \rangle) \mid \\
(\text{REPEAT} \langle \text{integer} \rangle \\
\quad (\text{plan-concept-expression}) \rangle) \mid \\
(\text{TEST} \langle \text{(state-concept)} \\
\quad \langle \text{plan-concept-expression} \rangle^+ \rangle) \mid \\
(\text{OR} \langle \text{plan-concept-expression} \rangle^+) \mid \\
(\text{SUBPLAN} \langle \text{symbol} \rangle)
\]

(\text{action-concept}) and (\text{state-concept}) refer to CLASSIC concepts subsumed by the concepts Action and State. Again, Action and State are pre-defined in CLASSIC by CLASP, as discussed in the previous section. Thus, a subset of the description-forming term language of CLASSIC is embedded in a CLASP description-forming plan language. Plan definitions restrict the type, the (conditional) presence, and sequential temporal ordering of action individuals in scenarios (plan individuals) described by the plan. We can also specify partial orders, using the operator OR to explicitly specify that any number of sequential descriptions are acceptable.

The following examples illustrate the interpretation of the constructs listed above:

- (SEQUENCE A B C): An action of type A is followed by an action of type B, which is followed by an action of type C.
- (LOOP A): Zero or more actions of type A.
- (REPEAT 7 A): Equivalent to (SEQUENCE A A A A A A A).
- (TEST (S1 A) (S2 B)): If the current state is of type S1, then action type A, else if state type S2 then action type B.
- (OR A B): Either action type A or type B.\(^9\)

\(^9\)Our algorithm for subsumption currently assumes that plans are expressed deterministically. Note that this means that the sets described by concepts A and B must be disjoint.
• (SUBPLAN Plan-Name): Syntactically insert Plan-Name's plan expression. Recursive definitions are not allowed.

The root of the plan taxonomy is the built-in CLASP concept Plan, where

(DEFINE-PLAN
 Plan
 (PRIMITIVE
   (AND Clasp-Thing
     (EXACTLY 1 INITIAL)
     (ALL INITIAL State)
     (EXACTLY 1 GOAL)
     (ALL GOAL State)
     (EXACTLY 1 PLAN-EXPRESSION)
     (ALL PLAN-EXPRESSION
      (LOOP Action)))))

Note that while the INITIAL and GOAL roles must be filled by a single state individual, the individual can be described by a State that is a complex conjunction of states.

As with terms, plan concepts can be specialized into subtypes, and organized into taxonomies:

(DEFINE-PLAN
 Plan-Subtype1
 (AND Plan
  (ALL PLAN-EXPRESSION
   (SEQUENCE
    Action-Subtype1
    Action-Subtype2
    (OR (SEQUENCE Action-Subtype3
         Action-Subtype4)
    Action))))

Informally, Plan-Subtype1 describes scenarios with three or four steps in which an action (individual) of subtype1 precedes an action of subtype2, which precedes either a sequence (an action of subtype3 followed by an action of subtype4), or a single action of any type. Of course, there must also be definitions for every action in this definition, defined using CLASSIC.

We can use the plan taxonomy to represent and organize descriptions of Definity™ 75/85 features (recall Section 2). For example, the following are informal CLASP definitions representing an abstract view of POTS (the default feature or "Plain Old Telephone Service"), as well as another plan used in the POTS definition:

\[\text{\textsuperscript{10}}\]

Unlike INITIAL and GOAL (filled by CLASSIC concepts), PLAN-EXPRESSION is restricted by fully specifying a plan concept, rather than by using restriction operators to specialize other concepts as in CLASSIC. Once specified, however, plan subsumption verifies that PLAN-EXPRESSION is indeed a restriction.
(DEFINE-PLAN
  Pots-Plan
  (AND
   Plan
   (ALL PLAN-EXPRESSION
    (SEQUENCE
     (SUBPLAN
      Originate-And-Dial-Plan)
     (TEST
      (Callee-On-Hook-State
       (SUBPLAN Terminate-Plan))
      (Callee-Off-Hook-State
       (SEQUENCE
        Non-Terminate-Act
        Caller-On-Hook-Act
        Disconnect-Act))))))

(DEFINE-PLAN
  Originate-And-Dial-Plan
  (AND
   Plan
   (ALL PLAN-EXPRESSION
    (SEQUENCE
     Caller-Off-Hook-Act
     Connect-Dialtone-Act
     Dial-Digits-Act))))

Again, there must also be definitions for every other action, state and plan in these definitions (defined using CLASSIC and CLASP, respectively). For example, the definitions of Connect-Dialtone-Act and Callee-Off-Hook-State in CLASSIC were presented earlier. Informally, Pots-Plan describes a plan in which the caller picks up a phone, gets a dialtone, and dials a callee. If the callee's phone is on-hook, the call goes through; if the callee's phone is off-hook, the caller gets a busy signal, hangs up, and is disconnected.

Note that with the exception of TEST and SUBPLAN, plan expressions built using the CLASP operators correspond to regular expressions. For example, Fig. 3 shows the finite state automaton which is equivalent to the plan expression of Plan-Subtype1, presented earlier. Here, the nodes represent states of the world, and the transitions correspond to the action operators that transform one state into another. CLASP, in fact, can transform all

![Fig. 3. Plan expressions as finite state automata.](image-url)
plan descriptions into regular expressions. First, CLASP can use the semantics underlying its action representation to replace all TEST operators with expressions involving OR operators and new action descriptions. This is because the preconditions of an action must describe the state of the world before an individual described by the action can be successfully executed. Similarly, in each (State Action) argument of a TEST operator, the state describes the state of the world that must be satisfied before an action individual can be described by the action. Thus, for example, (TEST (State1 Action1) (State2 Action2)) is equivalent to (OR Action1b Action2b), where Action1b specializes Action1 by specifying that the type restriction of PRECONDITION of Action1b is the conjunction of type State1 and the PRECONDITION type restriction of Action1. The construction of Action2b is similar. This construction will also cover the case when a complex plan expression takes the place of simple action descriptions such as Action1. This is because the complex expressions are “unbundled” into such simpler forms when constructing finite state machines, as discussed below. CLASP will thus construct and define such action descriptions, in order to replace all expressions involving the operator TEST with equivalent expressions involving the operator OR. As for SUBPLAN, recall that this construct was just a notational convenience. The ability to perform this transformation will have important implications with respect to plan subsumption. In particular, plan subsumption will be able to use results in automata theory to extend term subsumption.

Although plans are defined within CLASP, they are internally represented using CLASSIC. This can be done because CLASSIC provides hooks for storing information (e.g., via Common LISP types such as lists) that CLASSIC itself cannot meaningfully represent. CLASP constructs a finite automaton recognizing the plan expression, uses an array to represent this automaton, and stores the array using a CLASSIC role restricted to fillers of LISP type Array. Note that although “represented”, this information is outside the scope of the CLASSIC classification and subsumption processes. (In contrast, CLASSIC can be used to both represent and subsume the state information in the roles INITIAL and GOAL.) Instead, CLASP constructs a plan taxonomy using both the subsumption algorithms of CLASSIC and its own plan subsumption algorithms, as described below.

3.2.2. Scenarios

A CLASP plan individual is called a scenario. As in classical planners such as STRIPS [15], a plan individual corresponds to a sequence of actions that when executed in an initial state achieves a goal state. Like STRIPS, the temporal ordering of action sequences in scenarios must be total; in other words, CLASP scenarios are linear. (Recall from Section 3.1, however, that CLASP states are conjunctions of propositions and are thus simpler than those allowed in STRIPS.) A scenario is created in CLASP by asserting that it is described by (a specialization of) the concept Plan, by specifying the CLASSIC individuals that satisfy the restrictions on the plan’s initial state as well as goal state, and by specifying a scenario expression that is described by the plan’s plan expression. (The initial state is the state in which the scenario begins to execute, the goal state is the state that scenario execution achieves, and the actions are the ordered sequence of action individuals constituting the scenario.) In particular, scenario expressions are built from CLASSIC action individuals:
\(\text{scenario-expression} ::= ((\text{action-individual})*).\)

The following is an example of a scenario:

\[
\begin{align*}
\text{(CREATE-SCENARIO} \\
\text{pots-busy-scenario} \\
\text{(AND Plan} \\
\text{(FILLS INITIAL state-u1on-u2off)} \\
\text{(FILLS GOAL state-u1on)} \\
\text{(FILLS PLAN-EXPRESSION} \\
\text{(caller-off-hook-u1} \\
\text{connect-dialtone-on-u1} \\
\text{dial-digits-u1-to-u2} \\
\text{non-terminate-on-u2} \\
\text{caller-on-hook-u1} \\
\text{disconnect-u1))))}
\end{align*}
\]

With the callee off-hook in the initial state, this scenario represents the case where a user picks up the phone, the system generates a dialtone, the user dials the callee, the system generates a busy signal (the call failed to terminate), the user hangs up, and the system disconnects. More precisely, we define \text{pots-busy-scenario} to be a Plan whose INITIAL is filled by \text{state-u1on-u2off}, whose GOAL is filled by \text{state-u1on}, and whose PLAN-EXPRESSION is filled by a sequence of action individuals that begins with \text{caller-off-hook-u1} and continues through to \text{disconnect-u1}. The individuals used here, such as \text{state-u1on-u2off} (the initial state in which the caller is on-hook and the callee is off-hook) and actions such as \text{caller-off-hook-u1} (the caller goes off-hook, i.e., picks up the phone) are assumed to have been previously defined in CLASSIC. The following examples illustrate two such definitions:

\[
\begin{align*}
\text{(CREATE-IND} \\
\text{state-u1on-u2off} \\
\text{(AND State-U1on State-U2off))}
\end{align*}
\]

\[
\begin{align*}
\text{(CREATE-IND} \\
\text{connect-dialtone-on-u1} \\
\text{(AND Connect-Dialtone-Act} \\
\text{(FILLS ACTOR switching-system)} \\
\text{(FILLS PRECONDITION state-u1off-idle))))
\end{align*}
\]

When a scenario is created, CLASP confirms that the given sequence of actions will indeed transform the specified initial state into the goal state (i.e., that the scenario is well formed). During this process any unspecified intermediate states (the fillers of the precondition and goal roles of each action individual) are inferred, and any partially specified intermediate states are completed, using the STRIPS rule, as discussed.

\[\text{For clarity, we reflect the type of an individual in its name. For example, state-u1on-u2off is an individual of a type that is a conjunction of two specializations of type State. Also, u1 refers to the caller and u2 to the callee, etc. Recall that roles of states are encoded propositionally.}\]
in Section 3.3.3. For example, such reasoning would compute the filler of the GOAL in connect-dialtone-on-ui. While such information is often computed during plan synthesis, typically it is not stored in the final plan. As will be seen below, CLASP needs such information to determine if a scenario is described by a plan. Work on plan reuse [20] has shown a similar need for the maintenance of such information.

Scenarios are described by (are instances of) plans. A scenario described by a plan is a member of the class (the set of scenarios) corresponding to the plan. Intuitively, a scenario is an instance of a plan if the temporal and conditional restrictions used to define the plan concept are satisfied in the scenario. As with plans, CLASSIC can represent but not classify scenarios. In particular while CLASSIC can determine if the scenario fillers of INITIAL and GOAL meet the plan's restrictions, it cannot determine this with respect to PLAN-EXPRESSION. As will be seen, CLASP determines if a sequence of action individuals is described by a plan expression (a “grammar” of action descriptions) by parsing, in conjunction with term subsumption. For example, CLASP will determine that pots-busy-scenario is described by Pots-Plan.

3.3. CLASP inference

Terminological knowledge representation systems are often distinguished from object-oriented programming systems by the fact that they provide subsumption and classification as well as inheritance inferences.

As discussed above, subsumption of plans and scenarios is outside the scope of CLASSIC and is instead performed by CLASP. In particular, CLASP subsumption creates a plan hierarchy within a CLASSIC knowledge base. In this section, we present the algorithms for computing terminological plan inferences, as well as algorithms for computing inferences more typical of planning rather than knowledge representation systems. As we will see in Section 4, terminological plan reasoning supports retrieval of scenarios given incomplete and abstract queries of plan descriptions.

3.3.1. Subsumption of scenarios

In this section we present the algorithm for scenario subsumption, which computes whether a plan describes a scenario. As in CLASSIC, we use the term “subsumption” rather than the term “recognition” to describe this inference. We will also say, informally, that a plan P describes a scenario s (or that s satisfies P) when P subsumes s. Scenario classification uses scenario subsumption to determine all plans that a scenario satisfies. Scenario subsumption enables CLASP to explicitly assert CLASSIC instance relationships between plans and scenarios. Intuitively, a plan P is satisfied by a scenario s if the class of scenarios described by P includes s, that is, if the restrictions defining P are satisfied by s. Recall that every plan has restrictions concerning the roles INITIAL, GOAL, and PLAN-EXPRESSION.

However, the role that performing terminological inferences plays with respect to designing knowledge representation languages is subject to debate. For example, Doyle and Patil [13] argue against restricting knowledge representation languages in order to support efficient subsumption, as was done in CLASSIC.
Let $t$-subsumes and $i$-subsumes refer to term and instance subsumption, respectively, as supported in current terminological systems such as CLASSIC. Also, let $s$-subsumes and $p$-subsumes refer to two new inferences, namely CLASP subsumption between PLAN-DESCRIPTION descriptions and scenarios, and CLASP subsumption between PLAN-DESCRIPTION descriptions, respectively. Then, since the restrictions regarding INITIAL and GOAL are CLASSIC restrictions, $s$ satisfies $P$ if INITIAL and GOAL of $s$ are $i$-subsumed by INITIAL and GOAL of $P$, and if PLAN-DESCRIPTION of $s$ is $s$-subsumed by PLAN-DESCRIPTION of $P$.

Informally, a plan expression of a scenario (call it $s$-exp) is $s$-subsumed by that of a plan (call it $P$-exp) if the action individuals of $s$-exp are $i$-subsumed by the action descriptions that constitute $P$-exp subject to the temporal and conditional restrictions specified by the CLASP operators of $P$-exp. More formally, $s$-exp is $s$-subsumed by $P$-exp if $s$-exp is a string in the language defined by $P$-exp. Let $\Sigma$ be the alphabet consisting of all concepts of type Action, $\Sigma_i$ the set of action individuals, and $\Sigma_i^*$ all strings of action individuals, i.e., strings of individuals where each individual is described by a symbol of $\Sigma$. A scenario expression is an element of $\Sigma_i^*$. A plan expression over $\Sigma$ denotes a subset of $\Sigma_i^*$, namely the set of strings from $\Sigma_i^*$ in the language generated by the plan expression. P-exp $s$-subsumes $s$-exp if $s$-exp is in the subset of $\Sigma_i^*$ denoted by $P$-exp.

For example, assume Action-Subtype1 $i$-subsumes $\text{action1}$, and Action-Subtype3 $i$-subsumes $\text{action3}$. Then, with $P$-exp equal to

\[(\text{SEQUENCE (LOOP (OR Action-Subtype1 Action-Subtype2)) Action-Subtype3})\]

and $s$-exp equal to

\[\text{(action1 action1 action3)},\]

$P$-exp $s$-subsumes $s$-exp. In contrast, if $s$-exp equals (action1), $P$-exp does not $s$-subsume $s$-exp. Every $s$-exp is $s$-subsumed by Plan's $P$-exp, (LOOP Action).

As an example in the domain of software switching, with $P$-exp equal to

\[(\text{SEQUENCE (OR Caller-On-Hook-Act Callee-On-Hook-Act) Hangup-Act})\]

and $s$-exp equal to

\[\text{(caller-on-hook1 busy-hangup10)},\]

---

13 An orthogonal plan taxonomy could also be organized via goals using $g$-subsumption. Informally, one plan $g$-subsumes another if the fillers of the GOAL and INITIAL roles satisfy simple term subsumption relationships; $g$-subsumption thus ignores any relationships (i.e., $s$-subsumption and $p$-subsumption) among plan expressions, and simply checks the conditions in the world before and after the execution of the plan.

14 Note that if states were expressed using the full expressive power of predicate calculus (as in STRIPS), general theorem proving would be needed to determine subsumption among the fillers of the INITIAL and GOAL roles.
P-exp s-subsumes s-exp assuming Caller-On-Hook-Act i-subsumes caller-on-hook1 and Hangup-Act i-subsumes busy-hangup10. Note that the use of i-subsumption enables the recognition of action individuals that are directly as well as abstractly described by the action terms in P-exp. In contrast, the above plan expression would not s-subsume the scenario expression (busy-hangup10).

Since CLASP plan expressions can be transformed into regular expressions, s-subsumption can be efficiently implemented. To test whether s-exp is s-subsumed by P-exp, CLASP tests whether s-exp is accepted by the finite automaton recognizing the language denoted by P-exp. (CLASP builds a finite automaton whenever a plan is defined.) Because each transition in the finite automaton corresponds to a subsumption rather than equality check, we call our automata extended finite automata (EFA). The particular pattern matching algorithm used in CLASP is $O(mn) i$-subsumptions in the worst case, where $m$ is the size of the finite state machine equivalent of the plan and $n$ is the number of action individuals in the scenario [33]. The complexity of i-subsumption will depend on the particular terminological model used. In CLASSIC, the complexity of determining whether an individual satisfies a description (no embedded defined concepts) is unknown, while determining whether an individual matches a concept is believed to be NP-hard or NP-complete [29]. However, if the plans and scenarios are constructed from a stable CLASSIC knowledge base of action concepts and instances, CLASSIC can be used to pre-compute and cache all the i-subsumptions between concepts and individuals in a taxonomy, for later use by CLASP. Using i-subsumption results that are computed and cached "off-line" would allow the complexity of s-subsumption to be simply $O(mn)$. Full details of the algorithms for translating plan expressions into EFA machines and for performing scenario subsumption are described in [10].

Again, we emphasize here that the CLASP algorithm performs i-subsumption rather than equality checking between action concepts in $\Sigma$ and action individuals in $\Sigma_i$. This integration of term subsumption with regular expression processing provides a powerful facility for retrieving scenarios using incomplete and abstract plan descriptions.

3.3.2. Subsumption of plans

Plan classification is the determination of all plans that are more general and all plans that are more specific than a given plan. Plan classification organizes plans (classes) into a taxonomy according to the subset relation. Plan classification is based on plan subsumption, determination of whether one plan description is more general than another. Plan subsumption enables CLASP to explicitly assert CLASSIC subtype relationships between plans and plans. As we will see in Section 4, plan classification supports a useful class of queries. A plan $P$ is more general than a plan $Q$ if any scenario that satisfies $Q$ necessarily also satisfies $P$. Because states are represented using CLASSIC, t-subsumption can be used to determine generality for descriptions filling the plan roles INITIAL and GOAL.\footnote{Again, note that if states were expressed using the full expressive power of predicate calculus (as in STRIPS), general theorem proving rather than the term subsumption inference provided by CLASSIC would be needed to determine subsumption among states.} CLASP, however, must provide a new inference which we call
\textit{p-subsumption} to determine \textit{PLAN-EXPRESSION} generality. Thus, a plan description plan subsumes another plan description, if

\begin{align*}
\text{INITIAL}(P) &\text{ t-subsumes } \text{INITIAL}(Q), \\
\text{GOAL}(P) &\text{ t-subsumes } \text{GOAL}(Q), \text{ and} \\
\text{PLAN-EXPRESSION}(P) &\text{ p-subsumes } \text{PLAN-EXPRESSION}(Q).
\end{align*}

While s-subsumption compares "grammars" and "strings", p-subsumption compares two grammars.

For plan expressions PE1 and PE2, PE1 \textit{p-subsumes} PE2 if the language described by PE1 is necessarily a superset of that described by PE2. If L1 is the set of scenarios satisfying PE1, and L2 the set satisfying PE2, PE2 is p-subsumed by PE1 if L2 \subseteq L1. For example, given action descriptions A, B, and C, where A only t-subsumes C,

\[(\text{SEQUENCE (LOOP A)} )
\quad (\text{OR B A})\]

p-subsumes \((\text{SEQUENCE A C B})\) but does not p-subsume \((\text{SEQUENCE B A})\). The plan expression of the root of the plan taxonomy, \((\text{LOOP Action})\), p-subsumes the plan expression of every plan description.

p-subsumption can be understood in terms of an extension of regular expression subsumption. Given two plan expressions PE1 and PE2, we compute the equivalent deterministic extended finite automata, \(EFA^1\) and \(EFA^2\), and their Cartesian product \(EFA^X\). The states of \(EFA^X\) are ordered pairs of the form \((s_1^j, s_2^j)\), where \(s_i^j\) is one of the states of the machine \(EFA^j, j = 1, 2\); \(EFA^X\) would reach state \((s_1^j, s_2^j)\) after scanning through a scenario \(S\) just in case the machine \(EFA^j\) would be in state \(s_i^j, j = 1, 2,\) after scanning through the same scenario \(S\).

The Cartesian product machine helps us determine if all the scenarios accepted by one EFA are also accepted by the other (i.e., if one EFA p-subsumes the other). This is accomplished by looking in the product machine for states where one of the machines accepts and the other doesn't, as well as states where both accept. If a state of the form \(\langle \text{accept}, \text{non-accept}\rangle\) occurs, that means there is a scenario where \(EFA^1\) accepts and \(EFA^2\) non-accepts. Now, if there are states of the form \(\langle \text{accept, accept}\rangle, \langle \text{non-accept, accept}\rangle, \) and \(\text{no states of the form } \langle \text{accept, non-accept}\rangle, \text{ then clearly, all scenarios accepted by } EFA^1 \text{ are also accepted by } EFA^2 \) (and \(EFA^2\) accepts additional scenarios); thus, in this case, PE1 is p-subsumed by PE2. Likewise, if the product machine contains states of the form \(\langle \text{accept, accept}\rangle, \langle \text{accept, non-accept}\rangle, \) and \(\text{no } \langle \text{non-accept, accept}\rangle \) states, then the first plan subsumes the second. If there are only states of the form \(\langle \text{accept, accept}\rangle, \) the two plans accept the same language. We can also distinguish between cases where there are not subsumption relationships between the two plans. If there are \(\langle \text{accept, non-accept}\rangle, \langle \text{non-accept, accept}\rangle, \) and \(\langle \text{accept, accept}\rangle \) states, PE1 intersects PE2. That is, while neither plan subsumes the other, it is possible to have a scenario that could be described by both plans. Otherwise, if there are no \(\langle \text{accept, accept}\rangle\) states, the two plans are disjoint, as there can never be a scenario that is described by both plans. For example, consider the following plan expressions, where concepts A and B are disjoint, and concept A1 specializes A:
Fig. 4. Constructing the Cartesian product machine.

PI: (LOOP A)
P2: (OR A1 B)
P3: (SEQUENCE A1 A1)

The results of p-subsumption will determine that PI intersects P2, PI subsumes P3, and P2 is disjoint from P3.

The construction of $EFA^x$, however, is not simple. Recall that the transitions in $EFA_{1,2}$ involve subsumption tests, not equality. Fig. 4 illustrates the complexity of computing the product when transitions involve subsumption tests. (a) and (b) of the figure each show a sample machine consisting of one transition, corresponding to the plan expressions (SEQUENCE A) and (SEQUENCE B), respectively. (c) shows the corresponding product machine assuming the transitions are based on the standard equality tests (note that the states are now pairs of individual states in the original machines). (d), in contrast, shows the extended cross-product machine if the transitions are instead interpreted as subsumption tests. In particular, here A and B denote intersecting CLASSIC action descriptions. Thus, when constructing the cross-product machine, we must ascertain relationships between the concepts representing the transitions of each machine (i.e., does a concept contain, equal, intersect, or not intersect with the other machine’s concepts), in order to generate all viable transitions in the cross-product machine.\[10 By examining the machine in (d), we can determine the p-subsumption relationship that the two plan expressions intersect, since there are states of the form $\langle$accept, non-accept$\rangle$ ($\langle2, R\rangle$), $\langle$non-accept, accept$\rangle$ ($\langle R, 4\rangle$), and $\langle$accept, accept$\rangle$ ($\langle2, 4\rangle$).

In Fig. 5, we show a full cross-product construction. The EFA in (c) is a cross-product of the machines in (a) and (b). The machine in (a) corresponds to the plan expression

\[
\text{(SEQUENCE A} \\
\text{(LOOP B) C)}
\]

while the machine in (b) corresponds to the plan expression

\[^{16} \text{CLASSIC provides tests for subsumption, disjointness and equality between concepts; with these, the meaningfulness of action concepts such as A & B, A & (not B) can be checked for the actions A and B. Though CLASSIC does not support negation, we construct these concepts merely to verify that the Cartesian product machines can reach certain states.} \]
We assume here that A, B, C are pairwise disjoint, and B t-subsumes D. The reader is encouraged to use the simple example in Fig. 4 as a guide and follow through the construction of the cross-product in Fig. 5. Since there are no states of the form \((\text{accept}, \text{accept})\) (that is, no state \((3, 3)\)), p-subsumption will return that the two plan expressions are disjoint.

As discussed, p-subsumption uses CLASSIC t-subsumption from concepts to concepts. t-subsumption for descriptions (no embedded defined concepts) is polynomial, while t-subsumption allowing defined concepts is believed to be NP-hard or NP-complete [29]. As with i-subsumption between concepts and individuals, if such results are cached, t-subsumption is a constant time operation once computed. The subsumption of regular expressions is P-SPACE hard [42]. The intractability of this problem arises from the fact that regular expressions and their equivalent non-deterministic finite state machines are very compact representations. In fact, if they are converted to their equivalent deterministic finite state machines (leading to an exponential increase in the size of the machines) the subsumption can be done in polynomial time. While computing p-subsumption we convert the CLASP plan expressions directly into finite state machines, prior to computing the cross-product. In the LASSIE domain, we find that the resulting finite state machines are generally quite small, and in practice, p-subsumption rarely takes
more than a few hundred milliseconds on SparcStation-1 class machine. (It should also be noted, as before, that CLASSIC t-subsumptions were previously computed and cached; they were essentially constant time operations.) A full ML pseudo-code description of the plan subsumption algorithm is presented in [10].

3.3.3. Action-based reasoning about scenarios

In addition to supporting terminological inferences, namely plan and scenario subsumption, CLASP also performs reasoning more typical of planning than of knowledge representation systems. In particular, when a scenario is created, CLASP uses the STRIPS [15] rule to compute the state individuals that would result after performance of every action in the scenario. Informally, the STRIPS rule specifies that when an action is applied in a particular world state, the new world state satisfies every state description in the action's ADD-LIST and every previously satisfied state description not in the DELETE-LIST.

If all intermediate states (i.e., the fillers of the PRECONDITION and GOAL roles of every action in a scenario) are fully specified when a scenario is created, CLASP uses this type of reasoning to just verify that the scenario indeed transforms the plan's initial state (the filler of the plan role INITIAL) into the plan's goal state (the filler of the plan role GOAL). However, if complete information is not specified when the scenario is created, CLASP will also assert the information in the form of appropriate fillers for the PRECONDITION and GOAL roles of the action individuals. As discussed above, CLASP needs complete state information to transform plan expressions into regular expressions, and to perform both plan and scenario subsumption. As will be seen in the next section, state individuals also provide further information for scenario retrieval.

Let s be a scenario of plan type P. Let the plan expression of s be the sequence of action individuals \( \langle \text{action}_1 \ldots \text{action}_n \rangle \), where each \( \text{action}_i \) is of type \( \text{Action}_i \). Recall from Section 3.1 that, since the state taxonomy is built using only the operator AND, every state description can be reduced to a simple conjunction of other state descriptions. There are \( n + 1 \) state individuals, denoting the states preceding and following the execution of each action individual in the scenario. Their relationship to the action individuals \( \text{action}_i \) is indicated below. For \( i \) from 1 to \( n \):

\[
\text{state}_i = \text{filler of the PRECONDITION slot of } \text{action}_i,
\]
\[
\text{state}_{i+1} = \text{filler of the GOAL slot of } \text{action}_i.
\]

There are also two additional relationships with slots of the scenarios:

\[
\text{state}_1 = \text{filler of the INITIAL slot of scenario } s,
\]
\[
\text{state}_{n+1} = \text{filler of the GOAL slot of scenario } s.
\]

Then, given \( \text{action}_i \) and individual \( \text{state}_i \), which instantiates \( \text{State}_i \)—a state represented by a conjunction of state concepts, we can compute \( \text{State}_{i+1} \) and create \( \text{state}_{i+1} \) as its instantiation. We define \( \text{State}_{i+1} \) inductively from \( \text{State}_i \), as defined schematically below (bear in mind that the \( \text{State}_i \), as well as the add and delete lists, are conjunctions of State concepts):
Once computed, CLASP can create a new individual \( \text{state}_{i+1} \) of type \( \text{State}_{i+1} \) within CLASSIC, and assert that it fills the GOAL and PRECONDITION roles of the actions \( \text{action}_i \) and \( \text{action}_{i+1} \), respectively.

As an example, let us apply the above processing to pots-busy-scenario, presented in Section 3.2.2. Recall that \( \text{state-\!u!on-\!u\!2 \!off} \) (corresponding to \( \text{state}_1 \) in the above formulas) is described by (AND \( \text{State-\!u!on} \) \( \text{State-\!u\!2 \!off} \)) (corresponding to \( \text{State}_1 \)). Also, since \( \text{action}_1 \) corresponds to caller-off-hook-\( \text{u\!1} \), \( \text{Action}_1 \) is the action concept Caller-Off-\( \text{Hook} \). Assume that the fillers of ADD-LIST and DELETE-LIST of \( \text{Action}_1 \) are restricted to (AND \( \text{Idle-State} \) \( \text{State-\!u\!2 \!off} \)) and \( \text{State-\!u\!on} \), respectively. We can then use the above rule to compute that \( \text{state}_2 \) (the state of the world after caller-off-hook-\( \text{u\!1} \)) is described by \( \text{State}_2 \):

\[
(\text{AND} \text{Idle-State} \text{State-\!u\!1 \!off} \text{State-\!u\!2 \!off}).
\]

CLASP will either (1) verify that the individual specified as \( \text{state}_2 \) (if specified within an action individual) satisfies this type, or (2) use CLASSIC to create a new individual \( \text{state}_2 \) of type \( \text{State}_2 \), and assert that \( \text{state}_2 \) FILLS the GOAL of caller-off-hook-\( \text{u\!1} \) and FILLS the PRECONDITION of connect-dialtone-on-\( \text{u\!1} \) in pots-busy-scenario (again, assuming that these roles are not already filled). Note that the interactions produced by action-based reasoning may also lead to the discovery of incoherency in plans and scenarios.

3.4. Summary

This section has presented the plan representation and inference facilities found in CLASP. CLASP provides a representation language for defining plan descriptions as sequential, iterative, deterministic, and non-deterministic compositions of CLASSIC action descriptions. CLASP also allows the creation of scenario individuals as specific sequences of CLASSIC action individuals.

CLASP supports several inferences to compute information implicit in the plan representations. Scenario subsumption determines whether a plan describes a scenario. Plan subsumption determines whether one plan is more general than another plan. Finally, special purpose state analysis reasons with Action concepts and individuals, in order to provide the state information needed by the plan-based terminological (subsumption) inferences.

4. Using CLASP

This section illustrates the use of CLASP in enhancing subsumption-based retrieval systems. The examples illustrate the power of combining term subsumption with plan processing. To date we have used CLASP to represent and query a small set of feature descriptions and feature scenarios added to a LASSIE knowledge base, as will be described below. CLASP representation and subsumption is fully implemented in Common
Lisp and CLASSIC. A manual, along with example knowledge base specifications and trace outputs of CLASP, can be found in [23].

As we have seen, CLASP enables the representation of classes of temporal and conditional compositions of actions, supporting the representation of feature information in the CLASSIF domain. The same mechanisms for representing and organizing feature descriptions such as Pots-Plan (Section 3.2.1) also provide a very flexible method for retrieving feature scenarios such as pots-busy-scenario. In particular, definitions of plan descriptions containing action "wildcards" can be used to retrieve scenarios satisfying particular planning relationships. The set of scenarios retrieved are just those scenarios that are subsumed by the query plan description during CLASP scenario subsumption. For example, a plan description containing the following plan expression can be used to find all the contexts in which specializations of the CLASSIC description Connect-Dialtone-Act occur:

\[(\text{SEQUENCE} \ (\text{LOOP Action}) \ Connect-\text{Dialtone-Act} \ (\text{LOOP Action})).\]

Here, CLASP represents context while CLASSIC adds context-independent action abstraction. Plan descriptions can also be used to retrieve scenarios satisfying temporal context relationships:

\[(\text{SEQUENCE} \ (\text{LOOP Action}) \ Caller-\text{On-Hook-Act} \ Callee-\text{On-Hook-Act}).\]

Here CLASP is used to specify that a caller goes on-hook immediately before a callee, and that all other actions in the scenario precede these actions.

We can also retrieve scenarios by specifying restrictions on intermediate states:

\[(\text{SEQUENCE} \ (\text{LOOP Action}) \ (\text{AND Action} \ (\text{ALL PRECONDITIONS Busy-State})) \ (\text{LOOP Action})).\]

This plan description is satisfied by any scenario in which a phone is busy at some point; the retrieval is based on information about role fillers of action individuals asserted into the knowledge base by the CLASP state computations.

Finally, plan subsumption can support retrieval of feature descriptions rather than scenarios. This capability could be used to determine if any existing features handle a target behavioral description. For example, plan classification could determine that a call forwarding plan description describes all plans in which a user dials the number of one phone, but another phone actually rings.

The capabilities supporting feature processing can also be used to flexibly retrieve test scripts. Suppose a tester wanted to run scripts to ensure that a particular process wrote a report to a log file every time any feature (call forwarding, call waiting, selective call reject) was enabled or disabled at a phone. Because many devices can be considered to be "phones" and not all features operate on these devices, finding all relevant scripts is
an extremely difficult task. With CLASP, however, the tester may issue a simple query:

\[
\text{SEQUENCE (LOOP Action)}
\]
\[
\text{(OR (AND Feature-Enable-Action}
\text{ (ALL HAS-OPERAND Phone))}
\]
\[
\text{(AND Feature-Disable-Action}
\text{ (ALL HAS-OPERAND Phone)))}
\]
\[
\text{(LOOP Action)).}
\]

Again, we see how scenario and term subsumption can be combined to support retrieval given incomplete and abstract descriptions. This query also illustrates how the taxonomic reasoning of CLASSIC can be combined with the regular expression facility of CLASP to help reduce the "cognitive load" on testers. For example, the tester does not have to remember that ISDN stations, ordinary handset telephones, incoming trunks, and many other devices are all considered to be "phones"; additionally, the user doesn't have to remember all the different features and how they are enabled.

CLASP can be used to query collections of message traces to identify those traces that show a certain behavior. For example, if we wanted to make sure that the switch didn't allow a particular incoming trunk to make an outgoing call, we could check that there were no message traces where an incoming trunk made a call into the system, and subsequently was able to dial out. This could be determined by constructing the following query:

\[
\text{(SEQUENCE (LOOP Action)}
\text{(AND Off-Hook-Action}}
\text{ (FILLS ACTOR incoming-trunk1))}
\]
\[
\text{(LOOP Action)}
\text{(AND Connect-Action}
\text{ (FILLS ACTOR incoming-trunk1)}
\text{ (ALL RECIPIENT Toll-Trunk))}
\]
\[
\text{(LOOP Action)).}
\]

If any answers are retrieved, then there are some violations.

5. Extending CLASP

Although the current representation language of CLASP can encode an interesting class of plans in the telephony domain, the language is not as general as many representations used in plan synthesis. The standard operators for composing plans not only include sequence and choice, but also iteration, recursion, and concurrency \[17\]. (However, many other plan languages besides CLASP do not include all of these constructs.) Some current models of plan representation are even more temporally complex \[2\]. All of these potential extensions pose challenges for the existing CLASP algorithms, as they require

\[17\text{If this were allowed, then it would be possible to dial into a PBX, and then dial out and make a free long-distance call.}\]
expressive power beyond that of the underlying finite state machine representation. In this section we illustrate in depth how the limitations of the underlying finite state machine representation causes one of the limitations of CLASP—an inability to specify conditionally terminated iterations—and discuss how CLASP can be extended to address this limitation. We focus here on a “WHILE-DO” type of operator; others such as “REPEAT-UNTIL” are straightforward extensions of this one.

5.1. The WHILE description-forming operator

Consider adding a description-forming operator called WHILE to the CLASP language:

\[(\text{WHILE } (\text{state-concept}) \ (\text{plan-concept-expression})).\]

For example, a plan concept expression such as

\[\text{PE0} \quad (\text{WHILE \ Callee-On-Hook-State \ Ring-Act})\]

would describe those scenarios consisting of a sequence of Ring-Act instances, where the PRECONDITION of each Ring-Act is an instance of Callee-On-Hook-State. Note that when embedded in other plan expressions, the WHILE is context sensitive. In particular, the Ring-Act instances consumed by the WHILE must persist as long as the preceding state (i.e., the instance's PRECONDITION) satisfies Callee-On-Hook-State. Thus, once the WHILE is terminated, the immediately following action individual cannot have a PRECONDITION that satisfies Callee-On-Hook-State, even if the action is not an instance of Ring-Act.

To make this point clear, let us contrast the meaning of the following two plan expressions:

\[\text{PE1} \quad (\text{SEQUENCE} \ (\text{LOOP} \ (\text{TEST} \ (\text{Callee-On-Hook-State \ Ring-Act}))) \ \text{Connect-User-Act}),\]
\[\text{PE2} \quad (\text{SEQUENCE} \ (\text{WHILE \ Callee-On-Hook-State \ Ring-Act}) \ \text{Connect-User-Act}).\]

In particular, let us define a scenario expression se1 that is described by both PE1 and PE2, and a scenario expression se2 that is described by PE1 but not by PE2:

\[\text{se1} \quad \text{(u2-on-hook-ring \ u2-on-hook-ring \ u2-off-hook-connect-user)},\]
\[\text{se2} \quad \text{(u2-on-hook-ring \ u2-on-hook-ring \ u2-off-hook-connect-user}).\]

The scenario expressions refer to three action individuals: u2-on-hook-ring, u2-on-hook-connect-user, and u2-off-hook-connect-user. The first action individual is an instance of Ring-Act, and the second and third are instances of the (disjoint) concept Connect-User-Act. The first two individuals have their PRECONDITION role filled by an individual u2-on-hook, which is an instance of Callee-On-Hook-State. The PRECONDITION of u2-off-hook-connect-user is filled by u2-off-hook, an instance of Callee-Off-Hook-State.

First consider se1. Since the two instances of Ring-Act each have preconditions satisfying Callee-On-Hook-State, these instances satisfy the LOOP portion of PE1. Since
the last action instance satisfies Connect-User-Act—the last act in the SEQUENCE—the scenario is described by PE1. The scenario expression se1 also satisfies PE2. Since the two instances of Ring-Act each have preconditions satisfying Callee-On-Hook-State, they are described by the WHILE loop. Since the immediately following instance of Connect-User-Act has a precondition that is an instance of Callee-Off-Hook-State, this instance both confirms the termination of the WHILE loop, and satisfies the Connect-User-Act ending the SEQUENCE.

In contrast, consider se2. This scenario is described by PE1 (following the same explanation as given for se1), but is not described by PE2. This is because

1. all action individuals described by the WHILE must both be instances of Ring-Act and have preconditions satisfying Callee-On-Hook-State, and
2. the WHILE cannot terminate unless the next action individual in the scenario has a precondition that does not satisfy Callee-On-Hook-State.

In se2, an instance of Connect-User-Act follows the two instances of Ring-Act. This third instance cannot be described by the WHILE, as it is not a Ring-Act. However, since its precondition still satisfies Callee-On-Hook-State, the WHILE loop can also not terminate after just describing the instances of Ring-Act. Thus, se2 does not satisfy PE2, because the WHILE cannot be satisfied.

5.2. Implementing WHILE loops as extended finite automata

Unlike TEST, we cannot simply implement WHILE by transforming a WHILE plan expression into a regular expression of action concepts (i.e., by translating WHILE into another plan expression that uses only the regular expression subset of CLASP constructs). In particular, the "look-ahead" involved in terminating the WHILE loop requires minor extensions to the basic CLASP framework. This section shows how to directly translate a WHILE plan expression into an extended finite automaton (EFA), in conjunction with an extension of the top level syntax for plan and scenario expressions.

The general schema for translating a WHILE loop of the form

\[(\text{WHILE } \text{StateX } \text{Plan-ExpressionX})\]

into an EFA is shown in Fig. 6.

The start state of this EFA is the intermediate (non-accepting) loop iteration node s1. This node has a transition into an EFA representing the plan expression

\[(\text{TEST } (\text{StateX } \text{Plan-ExpressionX}))\]

which itself has a transition back to s1. In addition, s1 has a transition into the reject node Reject, labeled

\[(\text{TEST } (\text{StateX } \text{Action}))\].

The intuition behind this translation is as follows. The loop portion of the EFA captures the non-context sensitive portion of the WHILE (recall PE1). When the WHILE chooses to terminate, the machine sees if the next state still satisfies the looping condition. If so, the transition from s1 to Reject is taken. Otherwise, the machine proceeds to the rest of the machine via the
(TEST ((NOT StateX) Fallowing Transition))

transitions. This transition (or set of transitions) will be taken when StateX is no longer true; at this point the loop termination condition is satisfied, and the remaining action instances in the scenario no longer have to match the plan expression in the WHILE loop body. We use a dotted line to indicate that the precondition of this transition needs to be "merged" with the next transition(s) generated from the portion of the larger embedding plan expression immediately after the WHILE loop. An example of the results of such a merging will be given below.

This translation works when the WHILE loop is embedded in a larger plan expression. However, when the WHILE loop constitutes an entire plan expression, or if it is the "final" construct in the plan expression, it does not work. We have no "following transitions" so there are no plan expressions to merge the (TEST ((NOT StateX) ...)) transition(s) into. To handle this problem, we modify CLASP plans and scenarios to always include a dummy END token that can be used to terminate WHILE loops when necessary. In particular, we modify the CLASP syntax to include the grammar rules shown in Fig. 7, where the grammars for plan-concept-expression and for scenario-concept-expression were defined in Sections 3.2.1 and 3.2.2, respectively.

The subscripts C and S are included in plans and scenarios respectively. END_C is an action concept whose only instance is the action individual END_S. The semantics of the "END"s are that they always occur at the end of every plan and scenario and indicate termination.\textsuperscript{19}

\textsuperscript{18} As noted earlier, some terminological systems (e.g., CLASSIC) can only represent limited forms of negation.

\textsuperscript{19} A similar problem, and alternative solutions, can be found in the literature on dynamic logic [18].
Subsumption of scenarios with WHILE loops

To see how s-subsumption with WHILE loops works, let us examine the processing of scenario se2 by a machine generated from PE2, using the above translation for the WHILE portion. To translate PE0 (the while of PE2) into an EFA, the machine in Fig. 6 is instantiated as follows. First, there is a loop transition from s1 to s1 labeled

\((\text{TEST} (\text{Callee-On-Hook-State Ring-Act}))\).

There is also a transition from s1 to Reject labeled

\((\text{TEST} (\text{Callee-On-Hook-State Action}))\).

Finally, there is a transition labeled

\((\text{TEST} ((\text{NOT Callee-On-Hook-State}) \text{Connect-User-Act})))\)

to a final (accepting) state s2, which corresponds to the "following transitions" shown in Fig. 6. (Since this example does not need to use the END token, we use the original grammar rules to simplify the presentation.)

The (non-deterministic) EFA begins in state s1. Since the first action individual u2-on-hook-ring can be described by both of the plan expressions labeling the outgoing transitions, either transition can be taken. So we can posit that whenever either transition is possible, we take the loop. Thus, the first action individual puts the machine back at s1, as does the second action individual. However, only the label on the transition to Reject can describe u2-on-hook-connect-user, the next action instance in the scenario. Thus, this path through the EFA leads to a rejecting state. Since all other possible paths through the EFA also lead to Reject, PE2 does not describe se2. In contrast, consider the processing of se1, where the last action individual is u2-off-hook-connect-user. While there are paths to Reject, there is also a path through the machine where the first action individual puts the machine back at s1, as does the second action individual. Finally, u2-off-hook-connect-user instantiates (TEST ((\text{NOT Callee-On-Hook-State}) \text{Connect-User-Act})) and the machine takes the transition to s2, and halts, accepting se2.

More generally, suppose that S is a scenario with n action instances (where \(S_i\) is the ith instance), and that S contains a subsequence that satisfies (WHILE StateX Plan-ExpressionX). Also suppose that starting at some position lb (loop begin), the filler of PRECONDITION slot of a subsequence of action individuals satisfies StateX. Furthermore, assume that the loop iterates lr (loop repeat) times, then terminates with a match on the scenario processed so far (i.e., there is path through the EFA that does not enter the reject state). Assume that the last action instance described by the WHILE is
at position $l_f$ (loop finish). Finally, assume that the position of the unconsumed action individual to be matched at the beginning of the $j$th iteration of the loop ($1 \leq j \leq l_r$) is given by $ls_j$ ($ls_1 = lb$, and $ls_{l_r} = l_f$).

Consider what happens as the EFA shown in Fig. 6 processes scenario $S$, starting at position $lb$. The machine is in state $s_1$, $j = 1$, and by the assumptions above, the precondition of $S_{lb}$ is described by $\text{StateX}$. Since the machine is non-deterministic, to process the first action individual ($S_{lb}$), the state $s_1$ splits. After the first iteration successfully completes, the EFA will have consumed the action instances at positions $lb$ through $ls_1$, and can be in either of the two states $s_1$ and $\text{Reject}$. At the beginning of each subsequent loop iteration, the $s_1$ state is again split, as the different paths through the EFA are attempted. Thus, after $j$ successful iterations, the possible states of the EFA are:

\[ s_1, \text{Reject}, \text{Reject}, \ldots \quad (j \text{ times}). \]

At the end of $l_r$ iterations, the machine is looking at $S_{lf+1}$; recall from the assumptions above, that this action individual immediately follows the part of the scenario described by the loop. Thus, the machine can no longer take the transition through the loop back to $s_1$: in other words, $s_1$ is no longer split, and only the other transition in the EFA can be active. At this point, the possible states of the EFA are:

\[ s_1, \text{Reject}, \text{Reject}, \ldots \quad (l_r \text{ times}). \]

Now, if the current state (the precondition of $S_{lf+1}$) satisfies $\text{StateX}$, the above state set becomes

\[ \text{Reject}, \text{Reject}, \text{Reject}, \ldots \quad (l_r + 1 \text{ times}). \]

That is, if the state satisfies $\text{StateX}$, but the individuals in the scenario don’t match another iteration through the WHILE loop, the EFA representing the WHILE loop rejects the scenario. If the state doesn’t match $\text{StateX}$, the succeeding transition in the embedding EFA can be attempted. Of course, even if the embedded EFA rejects, other parts of the larger EFA (e.g., if the WHILE was inside an OR) may still have active transitions.

Finally, let us consider a plan expression consisting of just one WHILE loop. In particular, assume the scenario expression is $(u2-on-hook-ring)$ and the plan expression is $\text{PEO}$. By the new grammar rules shown in Fig. 7, the top level scenario expression is $(u2-on-hook-ring \text{ END})$ and the top level plan expression is $(\text{SEQUENCE (WHILE Callee-On-Hook-State Ring-Act) END})$. After the ring instance is processed, control returns to the embedding EFA—which was generated for the SEQUENCE. Here a transition labeled END is taken to an accepting state.

5.4. Subsumption of plans with WHILE loops

We have described how the EFA translation of WHILE works for scenario subsumption. Now let us turn to the computation of the subsumption inference between two plans, when plans are allowed to contain WHILE loops. Plan subsumption works using
the cross-product algorithm described earlier, except for one important difference. Recall that the plan subsumption algorithm made an assumption that the concepts labeling the transitions going out of a state were pairwise disjoint. This assumption is preserved by the translations of all CLASP constructs into EFAs, except for WHILE. In Fig. 6, notice that the label on the transition going to Reject is (StateX Action) (where Action is the most general action concept). This will certainly subsume the label(s) on the other transition(s) starting at s1 (generated during the EFA translation of (TEST (StateX Plan-ExpressionX))).

To remove this unacceptable non-determinism from a machine M, we simply take the cross-product of M with itself, exactly as we do in the case of the plan subsumption algorithm. In this algorithm, we "cross" every pair of transitions out of a state s in M. Each pair can potentially give rise to four transitions, corresponding to every possible boolean combination of the two transitions.

This removes the unacceptable non-determinism. The machine may be non-deterministic in the sense that it can be in several states at once, but no pair of transitions out of any one state will correspond to overlapping concepts. Now, the standard CLASP subsumption algorithm will again be applicable.

6. Related work

Taxonomic reasoning has largely been the concern of research in knowledge representation, particularly in KL-ONE-like systems. While plan reasoning has typically been the concern of fairly orthogonal research areas, namely plan recognition and plan synthesis, there are a few examples where abstraction in planning plays an important role.20

Several plan-based representation and reasoning systems have been concerned with the development of algorithms for plan subsumption, and the use of an existing terminological system to represent the building blocks of plans. In Wellman's plan synthesis work [42], plans are built from actions represented in the term subsumption language NIKL [37], and plan classes are organized into a subsumption taxonomy. However, the actual language for constructing plan descriptions is quite different (largely due to differences in domain); for example, Wellman's language is totally atemporal while CLASP allows the representation of sequence. Correspondingly, the algorithms for classifying and subsuming plan descriptions must differ. Furthermore, Wellman only represents and subsumes plan classes, while CLASP is also concerned with plan individuals. Wellman uses plan subsumption to provide a new approach to plan synthesis.

In the plan recognition work of Weida and Litman [40], plan decompositions are represented using temporal constraint networks, where the nodes are actions represented in the terminological language K-REP [27], and the arcs are disjunctions of qualitative temporal constraints [1]. More recently, Weida and Litman allow networks to also include equality and temporal metric constraints, and nodes to be represented using CLASSIC as well as K-REP [41]. Plan subsumption is performed by integrating term subsumption with techniques from the area of constraint satisfaction, and forms the basis

20 Plan decomposition hierarchies also play a role in several systems [121,321].
of a terminological approach to plan recognition. While the constraint network approach supports the representation of simultaneous actions and has a more expressive sequential representation than CLASP, it does not support action disjunction, action iteration, or the representation of states.

The RAT system [19] is built on top of the terminological system KRIS [4], and focuses on a language for representing complex states. Like CLASP, the representation of states is still expressed using the underlying terminological language, rather than predicate calculus. However, the state representation in RAT uses many of the constructs provided by KRIS and is thus more expressive than the state representation used in CLASP. In contrast, the language for composing plans is much richer in CLASP than in RAT, which only allows sequence. RAT is also concerned with checking plan feasibility and simulating execution.

There has also been work on directly extending terminological languages through the addition of new constructors, rather than building extensions on top of existing terminological systems. Borgida [6] presents techniques for extending terminological systems, and illustrates the techniques by reconstructing the plan subsumption reasoning developed in CLASP. To formally specify the extensions, Borgida presents axioms defining the semantics of CLASP using natural semantic rules of inference. Artale and Franconi [3] present the syntax and semantics of a terminological language that directly incorporates the qualitative temporal constraints that were used in the loosely integrated system of [40].

Several research projects have used existing terminological knowledge representation systems to directly represent and organize plans. Swartout and Neches [35] use NIKL to organize plans into a taxonomy by the semantics of the goals achieved (rather than the methods that are used to achieve the goals). In our terminology, the plan hierarchy is thus organized via g-subsumption, rather than by s-subsumption and p-subsumption as in CLASP. The COMET multimodal generation project [14] uses the terminological language LOOM [24] to "represent" sequential plans. COMET uses intuitive role-naming conventions (the first step of a plan is stored in a role called "step1", the second in a role called "step2", and so on) that are understood by the generation programs. This information is not understood, however, by the representation system during its subsumption processes. CLASP explicitly represents temporal and other planning relations; therefore, plan and scenario subsumption based on such sequencing information can be incorporated.

Plan taxonomies are also found in non-terminological approaches to plan recognition and synthesis. A plan abstraction hierarchy is central to the plan recognition work of Kautz [21]. However, in his taxonomy, plan nodes have no internal temporal or conditional structure. Also, no aspect of Kautz's representation is specified using a terminological system. Thus, his nodes have no terminological semantics and the suitability of his representation for computing terminological plan inferences is not of concern. In the field of plan synthesis, Tenenberg [36] uses a plan hierarchy to construct abstract plan solutions that constrain later search, where any abstract solution can always be specialized by choosing a specialization of each abstract plan step. Thus, while plans in Tenenberg's hierarchy are compositions of actions (like in CLASP), plans must always be structurally isomorphic across abstraction levels. In other words, the focus of such
ABSTRACTIONS [31] inspired work (in planning as well as in machine learning) is to use and generate abstraction hierarchies of action operators (based on elimination of their preconditions) [22]. Also, as with Kautz, this work is not at all integrated with or motivated by any concerns of work in terminological representation.

Terminological models have been integrated with other paradigms besides plans, namely rule-based systems. For example, in the work of [43], rules are composed from terms defined within the knowledge representation system LOOM, and a classification algorithm constructs a rule taxonomy based on the semantics of the left-hand side of such rules. The incorporation of term subsumption into a production system framework thus supports semantic pattern matching.

7. Conclusion

CLASP is a plan-based knowledge representation and reasoning system that combines terminological and regular expression processing. This combination enables CLASP to extend the expressive power of previous terminological approaches, by allowing the construction of plans from concepts corresponding to actions, using plan description-forming operators for choice, sequencing and looping. In this paper we presented the CLASP language for defining plan descriptions corresponding to classes and for creating plan individuals that satisfy such descriptions. Whenever a plan is defined, CLASP constructs the extended finite automaton that is equivalent to the plan's PLAN-EXPRESSION. As in terminological systems, descriptions and individuals are organized into taxonomies based on subsumption inferences. We have discussed subsumption inferences relevant to plans and scenarios—p-subsumption and s-subsumption—and have shown how such inferences are related to core notions of terminological and instance subsumption. In particular, p-subsumption and s-subsumption are computed by extending standard algorithms involving finite automata to the extended finite automata of CLASP, where transitions correspond to subsumption rather than equality checks. We have also shown how inferences specific to action representations can be used to support computation of the plan-based terminological inferences. The subsumption inferences provided by CLASP bring the many benefits of terminological reasoning to new application areas, namely those which require the representation of CLASP-like plans. We have motivated our work by demonstrating the importance of managing large collections of plans. In particular, we have used CLASP to provide a viable data model as well as a framework for representing and retrieving information in a software information system in the telephony domain.

Several extensions to CLASP would be particularly useful. We proposed an extension to CLASP that addressed one of the limitations on iteration imposed by the regular expression representation. Having done WHILE loops, REPEAT-UNTIL loops could be added to CLASP in a similar manner. Weida [39] presents some proposals for integrating simultaneous actions as well as more expressive sequential relationships. One might also add inheritance and an elementary notion of assertions to CLASP. Assertions would allow the definition of constraints on execution patterns and could be used during software debugging to check execution traces for anomalous behavior.
Acknowledgements

We would like to thank Ron Brachman and Henry Kautz for comments on a preliminary version of this paper, which appeared in [12]. Numerous discussions with Alex Borgida helped clarify many parts of this work, particularly the section on adding looping constructs. We are very grateful to Tony Weida for his comments on the current version. We would also like to thank Jim Piccarello and Ian Bruce for useful discussions regarding the application of knowledge representation to Definity™75/85. Finally, we would like to thank the anonymous reviewers for their useful suggestions and comments.

References

terminological logics, DFKI, Saarbrücken, Germany (1992).
[28] P.F. Patel-Schneider, Small can be beautiful in knowledge representation, in: Proceedings IEEE Workshop
[29] P.F. Patel-Schneider and A. Borguda, The (non-standard) treatment of individuals and user functions in
McGuinness, B. Nebel, A. Schmiedel and J. Yen, Term subsumption languages in knowledge
representation, AI Mag. 11 (2) (1990) 16–23.
[38] W. Wahlster, E. Andre, S. Bandyopadhyay, W. Graf and T. Rist, WIP: the coordinated generation of
multimodal presentations from a common representation, in: Proceedings International Workshop on
Computational Theories of Communication and their Applications: Problems and Prospects, Trentino
[40] R. Weida and D. Litman, Terminological reasoning with constraint networks and an application to plan
recognition, in: Proceedings Third International Conference on Principles of Knowledge Representation
[41] R. Weida and D. Litman, Subsumption and recognition of heterogeneous constraint networks, in: Proceedings Tenth IEEE Conference on Artificial Intelligence for Applications (CAIA-94), San Antonio,
TX (1994) 381–388.
MA (1988).