Taxonomic Plan Reasoning

Premkumar T. Devanbu Diane J. Litman
Artificial Intelligence Principles Research Department
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974*

October 11, 1991

Abstract

CLASP (CLAssification of Scenarios and Plans) is a knowledge representation system that extends the notion of subsumption from frame-based 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. We are using CLASP to store and retrieve information about feature specifications and test scripts in the context of a large software development project.

*Authors listed in alphabetic order only. Premkumar Devanbu is also with the Department of Computer Science, Rutgers University, New Brunswick, NJ. Diane Litman is currently with the Department of Computer Science, Columbia University, New York, NY 10027. Authors can be reached electronically via prem@research.att.com and litman@cs.columbia.edu.

1 Introduction

Terminological systems [21] are in the KL-ONE family [3] of knowledge representation languages, and provide representational support in many areas of Artificial Intelligence. Central to terminological approaches are the interpretation of frames as descriptions, the use of classification and term subsumption inferences[1] to organize frame taxonomies, and differentiation between terminological and assertional aspects of knowledge. A major limitation of current terminological 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 [28], expert systems [32], user interfaces [10, 35], plan synthesis [36], software information systems [7]). 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.

To motivate the need for a plan-based terminological representation system, consider the use of terminological representation systems in the area of software development. As the size, cost, and lifetime of software systems continue to grow, it becomes increasingly important to find ways to help programmers understand and maintain these systems [4, 6]. There are several practical systems [5, 31] that store and query information about control and data relationships using relational databases. There are also several tools [27] that are designed to present such information visually in a comprehensible and useful manner. However, the information stored in such systems is of a purely *syntactic* nature; information collection and retrieval are based on simple string matching. Such systems do not address the *semantics* of the specific domain of application, the architecture of the system, or how the architecture is designed to service the needs of the application domain. Fischer and Schneider [12] suggested that a large software project should instead use a knowledge base to collect and disseminate information about all aspects of the system under construction. For example, if information was stored in a terminological knowledge base, classification could be used as an organizational tool, and

[1] We assume the standard interpretation for these terms, as described in [2]. 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.

subsumption used to process queries semantically. The LASSIE software information system [7] in fact pursued this idea, using the terminological knowledge representation language CLASSIC [2] [2] to describe the architecture, domain model, and code of the AT&T Definity^{TM} 75/85, which is a scalable Private Branch Exchange (PBX) switching product.

By using a terminological language such as CLASSIC, LASSIE could organize descriptions into a taxonomy and 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 has representational needs beyond the capabilities of CLASSIC. Because terminological languages (including CLASSIC) have no meaningful way to represent or reason with *plans* (temporal compositions of actions), LASSIE can not be used to describe sequences of actions that achieve particular goals (e.g., Call Forwarding, as found in most modern telephone systems). Context-dependent, temporal, and other relationships cannot be captured. As we will see, plan-like knowledge is important to the entire range of activities associated with software development. In addition, action subsumption and classification do 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 knowledge representation system called CLASP (CLAsification 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. CLASP uses the semantics of these descriptions to associate plan descriptions with sets of plan individuals, and to organize plan descriptions into taxonomies based solely on terminological inferences.

Figure 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 individuals. Plan classes correspond to sets of plan instances that satisfy the descriptions. For example, *Plan-Class1* describes all plan instances in which an instance

[2] LASSIE was originally implemented using the terminological language KANDOR [20].

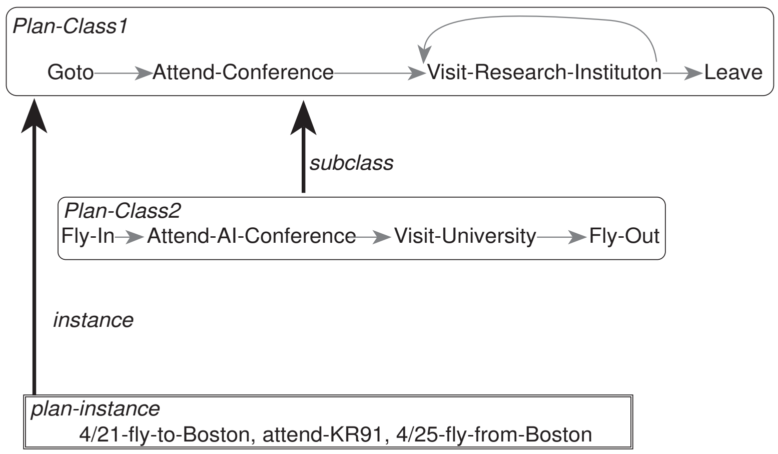


Figure 1: Terminological Plan Reasoning in CLASP

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 individuals shown. The CLASP terminological inferences use the semantics of the representations to organize plan classes and instances into taxonomies. For example, *plan-instance* is an *instance* of *Plan-Class1* because (1) the action individuals `4/21-fly-to-Boston`, `attend-KR91`, and `4/25-fly-from-Boston` 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 [2] terminological knowledge representation system, in order to allow CLASP to make use of CLASSIC'S well-defined subsumption inference. Thus, in the above example, CLASSIC term subsumption can be used to infer that `4/21-fly-to-Boston` is an instance of `Fly-In`, `4/21-fly-to-Boston` 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 plan-like knowledge in programming activity. 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 the programming domain. Finally, in Sections 5 and 6, we relate CLASP to existing research in taxonomic plan reasoning, and conclude with some thoughts on future directions.

2 Why Plan Knowledge?

Plans play a central role in many areas that provide applications for frame subsumption systems. In this paper we focus on the application area of software development to both motivate and illustrate the CLASP plan-based terminological system. Plan-like knowledge is pervasive in all phases of software development activity. As discussed above, the need to represent and reason with plan-like knowledge in the LASSIE software information system was in fact the original impetus for our development of CLASP.

The relationship between plans and software information has been explored most widely in the context of automatic and computer-assisted programming [23]. Plans allow the representation of the sequence of events in a program and the assertion of pre and post conditions that are applicable at various points. As another example, the PROUST[14] system uses a representation of programs in the form of plans to recognize and classify faults in student programs.

Empirical studies have shown that plan-like knowledge structures can serve as effective models for understanding human programming behavior [29, 30]. For example, plans play an important role in the recognition and comprehension of program fragments [29]; both correct comprehension as well as misunderstanding of programs are best explained by postulating the existence of plan-like cognitive structures. In [30], programmers perform a task involving changing a function that is part of a larger conceptual plan; when the entire plan is made available prior to the modification, performance improves significantly.

Besides the role played by planning knowledge in programming activity, there is also a need for plans during other phases of software development. In the domain of telephone switching software that was addressed in LASSIE, plan-like structures are particularly useful in representing *feature descriptions* 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.

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 extremely useful.

Plans can also be useful in explaining the behavior of distributed software systems, like the AT&T Definity^{TM} 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 is a "complement" to 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 framework shown in Figure 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, actions are the main building blocks of CLASP plans and scenarios. Since actions can be represented adequately in current terminological systems such as CLASSIC, the representation of actions in CLASSIC is briefly discussed here.

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,

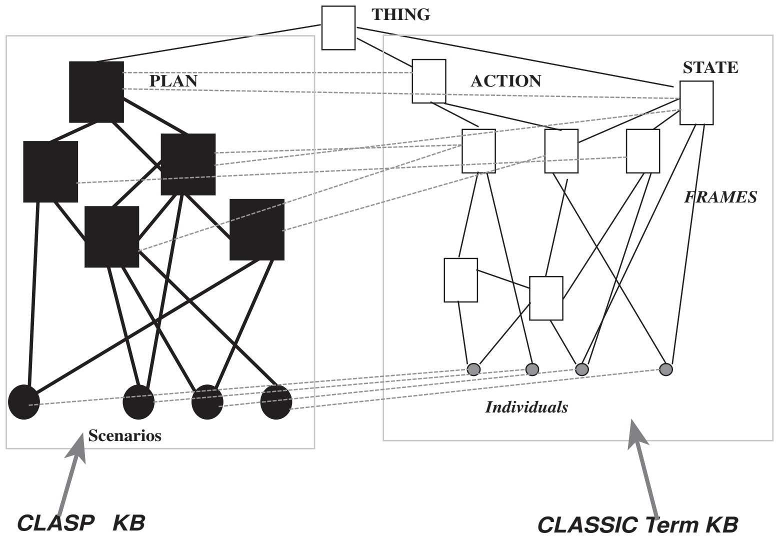


Figure 2: Building a Plan Taxonomy from a Term Taxonomy

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 a STRIPS action operator [11], where role restrictions specify the characteristics of the roles `ACTOR`, `PRECONDITION`, `ADD-LIST`, `DELETE-LIST`, and `GOAL`[3]. In particular, an action is defined thus:

(DEFINE-CONCEPT
    `Action`
    (PRIMITIVE
        (AND `Classic-Thing`

[3]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.

```
(AT-LEAST 1 ACTOR)
(ALL ACTOR Agent)
(ALL PRECONDITION State)
(ALL ADD-LIST State)
(ALL DELETE-LIST State)
(ALL GOAL State))))
```

The above definition states that "An `Action` is a `Classic-Thing`, with at least one `ACTOR`, all of whose `ACTOR`s are of type `Agent`, all of whose `PRECONDITION`s are of type `State`, all of whose `ADD-LIST`s are of type `State`, all of whose `DELETE-LIST`s are of type `State`, and all of whose `GOAL`s are of type `State`" (assuming the previous definition of the concepts `Classic-Thing`, `State` and `Agent` in CLASSIC). 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 is not necessary and sufficient, that is, that the concept cannot be automatically placed into a concept taxonomy via subsumption and classification inferences.

The concept `State` is also pre-defined by CLASP - as a primitive CLASSIC concept specializing `Classic-Thing`. Furthermore, CLASP allows `State`s to only be restricted using the CLASSIC description-forming operator AND. All `States` can thus be reduced to a simple conjunction of other `States`. This representation will facilitate STRIPS-like tracking of state information, as discussed in Section 3.3.3.

`Action`s can be restricted to define various specialized actions in the LASSIE domain. As we will see, these action 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-Act`s and `User-Act`s. `System-Act` is defined below:

```
(DEFINE-CONCEPT
   System-Act
   (AND Action
        (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:

```
(DEFINE-CONCEPT
    Connect-Dialtone-Act
    (AND System-Act
         (ALL PRECONDITION
              (AND Off-Hook-State
                   Idle-State))
         (ALL ADD-LIST Dialtone-State)
         (ALL DELETE-LIST Idle-State)
         (ALL GOAL
              (AND Off-Hook-State
                   Dialtone-State))))
```

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`.

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):

```
(CREATE-IND
    act1
    (AND System-Act
         (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`.

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 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 actions, 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`. Plan concept expressions are compositionally defined from action and state concepts using the plan description forming operators SEQUENCE, LOOP, REPEAT, TEST, OR, and SUBPLAN:

<plan-concept-expression> ::=
    <action-concept> |
    (SEQUENCE <plan-concept-expression>+) |
    (LOOP <plan-concept-expression>) |
    (REPEAT <integer>
            <plan-concept-expression>) |
    (TEST (<state-concept>
          <plan-concept-expression>)+) |
    (OR <plan-concept-expression>+) |
    (SUBPLAN <symbol>)

In other words, the description-forming term language of CLASSIC is embedded in a CLASP description-forming plan language. (<action-concept> and <state-concept> refer to CLASSIC concepts subsumed by the concepts `Action` and `State`. Recall that `Action` and `State` are pre-defined in CLASSIC by CLASP.) 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`[4].
- (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
                (ALL INITIAL State)
                (ALL GOAL State)
                (EXACTLY 1 PLAN-EXPRESSION)
                (ALL PLAN-EXPRESSION
                      (LOOP Action)))))
```

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 and AT-LEAST operators with the same number.

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

[4]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.

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 "**P**lain **O**ld **T**elephone **S**ervice"), as well as another plan used in the POTS definition:[5]

```
(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
```

[5]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.

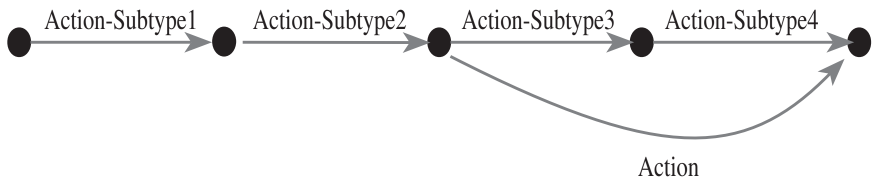


Figure 3: Plan Expressions as Finite State Automata

```
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 definition of `Connect-Dialtone-Act` in CLASSIC was 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, Figure 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 plan representations 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 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 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," the planning information is outside the scope of the CLASSIC classification and subsumption processes. Instead, CLASP constructs a plan taxonomy using 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 [11], 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*. A scenario is created in CLASP by asserting that it is described by (a specialization of) the concept `Plan`, by specifying the 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:

<scenario-expression> ::= (<action-individual>+).

The following is an example of a scenario:

```
(CREATE-SCENARIO
    pots-busy-scenario
    (AND Plan
        (FILLS INITIAL state-u1on-u2off)
        (FILLS GOAL state-u1on)
        (FILLS PLAN-EXPRESSION
            (caller-off-hook-u1
             connect-dialtone-on-u1
             dial-digits-u1-to-u2
             non-terminate-on-u2
             caller-on-hook-u1
             disconnect-u1)))).
```

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 `pots-busy-scenario` to be a `Plan` whose `INITIAL` is filled by `state-u1on-u2off`, whose `GOAL` is filled by `state-u1on`, and whose `PLAN-EXPRESSION` is filled by a sequence of action individuals that begins with `caller-off-hook-u1` and continues through to `disconnect-u1`. The individuals used here, such as `state-u1on-u2off` (the initial state in which the caller is on-hook and the callee is off hook) and actions such as `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[6]:

```
(CREATE-IND
   state-u1on-u2off
   (AND State-U1on State-U2off))

(CREATE-IND
   connect-dialtone-on-u1
   (AND Connect-Dialtone-Act
```

[6]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. Due to the inability to express complex role relationships in CLASSIC, in our domain we encode information about users using concepts rather than role restrictions.

```
(FILLS ACTOR switching-system)
(FILLS PRECONDITION state-u1off-idle)))
```

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, using the STRIPS rule, as discussed in Section 3.3.3. For example, such reasoning would compute the filler of the `GOAL` in `connect-dialtone-on-u1`. 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 [15] 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

Frame-based 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.[7] 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

[7]However, the role that performing terminological inferences plays with respect to designing knowledge representation languages is subject to debate. For example, Doyle and Patil [9] argue against restricting knowledge representation languages in order to support efficient subsumption, as was done in CLASSIC.

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 "realization" 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 `P` describes `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`.

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-EXPRESSION` restrictions and fillers, and CLASP subsumption between `PLAN-EXPRESSION` restrictions, respectively[8]. 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-EXPRESSION` of `s` is s-subsumed by `PLAN-EXPRESSION` 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 Σ be the alphabet consisting of all concepts of type `Action`, Σ_i the set of action individuals, and Σ_i^* all strings of action individuals, i.e., strings of individuals where

[8]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.

each individual is described by a symbol of Σ. A scenario expression is an element of Σ_i^*. A plan expression over Σ denotes a subset of Σ_i^*, namely the set of strings from Σ_i^* in the language generated by the plan expression. `P-exp` s-subsumes `s-exp` if `s-exp` is in the subset of Σ_i^* denoted by `P-exp`.

For example, assume `Action-Subtype1` i-subsumes `action1`, and `Action-Subtype3` i-subsumes `action3`. Then, with `P-exp` equal to

```
(SEQUENCE (LOOP (OR Action-Subtype1
                    Action-Subtype2))
          Action-Subtype3)
```

and `s-exp` equal to

```
(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

```
(SEQUENCE (OR Caller-On-Hook-Act
              Callee-On-Hook-Act)
          Hangup-Act)
```

and `s-exp` equal to

```
(caller-on-hook1 busy-hangup10),
```

`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 [26]. The complexity of i-subsumption will depend on the particular terminological model used. In CLASSIC, 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 [22]. 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 $O(mn)$.

Again, we emphasize here that the CLASP algorithm performs i-subsumption rather than equality checking between action concepts in Σ and action individuals in Σ_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. 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`. t-subsumption can be used to determine generality for descriptions filling the plan roles `INITIAL` and `GOAL`. CLASP, however, must provide a new inference which we call *p-subsumption* to determine `PLAN-EXPRESSION` generality. Thus, a plan description plan-subsumes another plan description, if

INITIAL(`P`) t-subsumes INITIAL(`Q`),
GOAL(`P`) t-subsumes GOAL(`Q`), and
PLAN-EXPRESSION(`P`) p-subsumes PLAN-EXPRESSION(`Q`).

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

For plan expressions `PE1` and `PE2`, `PE1` *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`,

(SEQUENCE (LOOP `A`) (OR `B A`))

p-subsumes (SEQUENCE `A C B`) but does not p-subsume (SEQUENCE `B A`). The plan expression of the root of the plan taxonomy, (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 automatons, EFA^1 and EFA^2, and their Cartesian product EFA^χ. The states of EFA^χ are ordered pairs of the form $< s_i^1, s_i^2 >$, where s_i^j is one of the states of the machine $EFA^j, j = 1, 2$; EFA^χ would reach state $< s_i^1, s_i^2 >$ 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 <*accept,non-accept* > occurs, that means there is a scenario where EFA^1 accepts and EFA^2 non-accepts. Now, if there are states of the form <*accept,accept* >, <*non-accept,accept* >, and *no* states of the form <*accept, non-accept* >, then clearly, *all* scenarios accepted by EFA^1 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 <*accept, accept* >, <*accept, non-accept* >, and no <*non-accept, accept* > states, then the first plan subsumes the second. If there are only states of the form <*accept, accept* >, 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 <*accept, non-accept* >, <*non-accept, accept* >, and <*accept, accept* > 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 <*accept, accept* > 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`:

P1: (LOOP `A`)

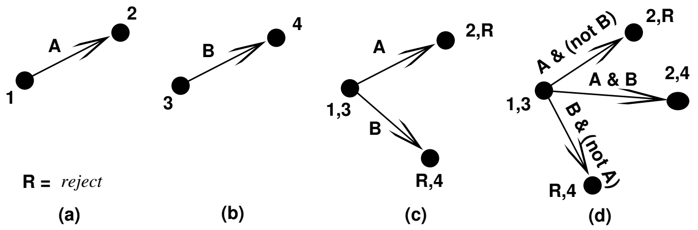


Figure 4: Constructing the Cartesian Product Machine

P2: (OR `A1 B`)
P3: (SEQUENCE `A1 A1`)

The results of p-subsumption will determine that P1 intersects P2, P1 subsumes P3, and P2 is disjoint from P3.

The construction of $EFA^{\times}$, however, is not simple. Recall that the transitions in $EFA^{1,2}$ involve subsumption tests, not equality. Figure 1 illustrates the complexity of computing the product when transitions involve subsumption tests. Portions (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. Portion (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). Portion (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. [9] By examining the machine in (d), we can deterimine the p-subsumption relationship that the two plan expressions intersect, since there are states of the form <*accept, non-accept* > (<2, R >), <*non-accept, accept* > (<R, 4 >), and <*accept, accept* > (<2, 4 >).

In Figure 5, we show a full cross product construction. The EFA in portion (c)

[9] 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.

is a cross product of the machines in (a) and (b). The machine in (a) corresponds to the plan-expression

```
(SEQUENCE A
          (LOOP B)
          C)
```

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

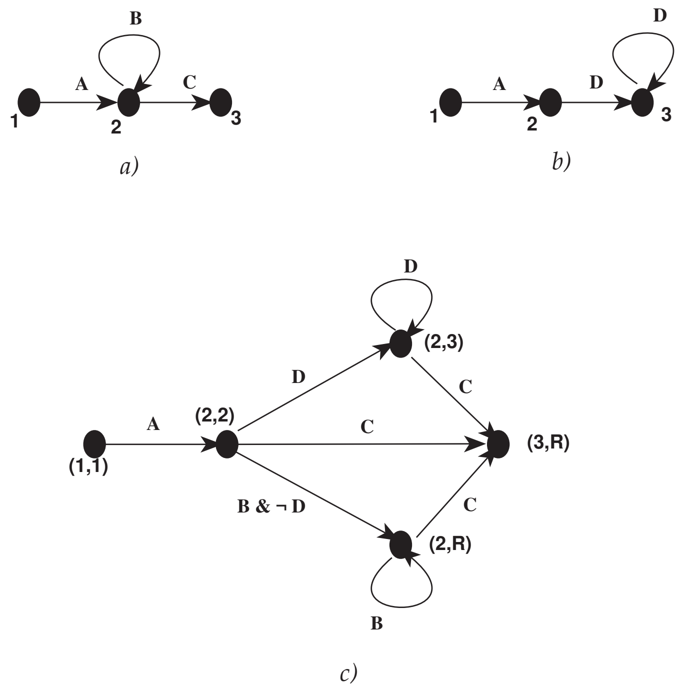
```
(SEQUENCE A
          D
          (LOOP D)).
```

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 Figure 4 as a guide and follow through the construction of the cross product in Figure 5. Since there are no states of the form <*accept, 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.[22] 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 [36]. 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. (It should also be noted, as before, that CLASSIC t-subsumptions were previously computed and cached; they were essentially constant time operations.)

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



a) b) c)

Figure 5: Full Cartesian Product of Two Machines (where `B` t-subsumes `D`)

representation systems. In particular, when a scenario is created, CLASP uses the STRIPS [11] 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 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 ($action_1 \ \ldots \ \ action_n$), where each $action_i$ is of type $Action_i$. 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. For i from 1 to n, let

$$\begin{aligned} state_1 &= \texttt{INITIAL(s)}, \\ state_i &= \texttt{PRECONDITION}(action_i), \\ state_{i+1} &= \texttt{GOAL}(action_i), \\ state_{n+1} &= \texttt{GOAL(s)} \end{aligned}$$

Then, given $action_i$ and individual $state_i$ of type $State_i$, we can compute $state_{i+1}$ of type $State_{i+1}$, such that

$$State_{i+1} = State_i \text{ - } \texttt{DELETE-LIST}(Action_i) + \texttt{ADD-LIST}(Action_i)$$

Once computed, CLASP can create a new individual $state_{i+1}$ of type $State_{i+1}$ within CLASSIC, and assert that it fills the `GOAL` and `PRECONDITION` roles of the actions $action_i$ and $action_{i+1}$, respectively.

As an example, let us apply the above processing to `pots-busy-scenario`, presented in Section 3.2.2. Recall that `state-u1on-u2off` (corresponding to $state_1$ in the above formulas) is described by (AND `State-U1on State-U2off`) (corresponding to $State_1$). Also, since $action_1$ corresponds to `caller-off-hook-u1`, $Action_1$ is the action concept `Caller-Off-Hook`. Assume that the fillers of `ADD-LIST` and `DELETE-LIST` of $Action_1$ are restricted to (AND `Idle-State State-U1off`) and `State-U1on`, respectively. We can then use the above rule to compute that $state_2$ (the state of the world after `caller-off-hook-u1`) is described by $State_2$:

(AND `Idle-State State-U1off State-U2off`).

CLASP will either (1) verify that the individual specified as $state_2$ (if specified within an action individual) satisfies this type, or (2) use CLASSIC to create a new individual $state_2$ of type $State_2$, and assert that $state_2$ FILLS the `GOAL` of `caller-off-hook-u1` and FILLS the `PRECONDITION` of `connect-dialtone-on-u1` in `pots-busy-scenario` (again, assuming that these roles are not already filled).

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 [18].

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 LASSIE 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:

```
(SEQUENCE (LOOP Action)
          Connect-Dialtone-Act
          (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:

```
(SEQUENCE (LOOP Action)
          Caller-On-Hook-Act
          Callee-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:

```
(SEQUENCE
   (LOOP Action)
   (AND Action
        (ALL PRECONDITIONS Busy-State))
   (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:

```
(SEQUENCE
   (LOOP Action)
   (OR (AND Feature-Enable-Action
              (ALL HAS-OPERAND Phone))
        (AND Feature-Disable-Action
              (ALL HAS-OPERAND Phone)))
   (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[10], we could check that there were no message traces where a incoming trunk made a call into the system, and subsequently was able to dial out. This could be determined by constructing the following query:

[10] 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.

```
(SEQUENCE
   (LOOP Action)
   (AND Off-Hook-Action
         (FILLS ACTOR incoming-trunk1))
   (LOOP Action)
   (AND Connect-Action
         (FILLS ACTOR incoming-trunk1)
         (ALL RECIPIENT Toll-Trunk))
   (LOOP Action)).
```

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

5 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[11].

A plan abstraction hierarchy is central to the plan recognition work of Kautz [16]. However, in his taxonomy, plan nodes are analogous to terms rather than structures. Also, no aspect of Kautz's representation is specified using a terminological system, and the suitability of his representation for computing terminological plan inferences is not of concern.

In the field of plan synthesis, Tenenberg [33] 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 ABSTRIPS [25] 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). [17] Also, as with Kautz, this work is not at all integrated with or motivated by any concerns of work in terminological representation.

[11] Plan decomposition hierarchies also play a role in several systems[24, 16].

The notion of abstraction is also central to Wellman's plan synthesis work [36], which is in fact the most similar approach to CLASP. Plans are built from actions represented in the term subsumption language NIKL [34], and plan classes are organized into a hierarchy based on a notion of subsumption. 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.

Several research projects have used existing terminological knowledge representation systems to directly represent and organize plans. Swartout and Neches [32] 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 [10] uses the terminological language LOOM [19] 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 provides a principled way to explicitly represent temporal and other planning relations within the representation system, so that plan and scenario subsumption based on information such as sequence can be designed and implemented.

Terminological models have been integrated with other paradigms besides plans, namely rule-based systems. For example, in the work of [37], 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.

6 Conclusion

CLASP is a *plan-based* knowledge representation and reasoning system that combines *terminological* and *regular expression* processing. In this paper we presented the CLASP language for defining plan descriptions corresponding to classes and for creating plan individuals that satisfy such descriptions. 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. We have also shown how inferences specific to action representations can be used to support computation of the plan-based terminological inferences. We have motived 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 software information systems.

Several extensions to CLASP would be particularly useful. Although the current representation language can encode an interesting class of plans in the telephony domain, the language is not as general as many representations used in plan synthesis. For example, the standard operators for composing plans not only include sequence and choice, but also iteration, recursion, and concurrency [13]. Some current models of plan representation are in fact extremely temporally complex [1]. However, to extend the expressive power (for example, allowing the representation of parallel actions), new algorithms for plan subsumption would be needed. 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 [8]. We would also like to thank Jim Piccarello and Ian Bruce for useful discussions regarding the application of knowledge representation to Definity 75/85.

References

[1] James F. Allen. Planning as temporal reasoning. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 3–14, Cambridge, MA, 1991.

[2] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with classic: When and how to use a kl-one-like language. In J. Sowa, editor, *Formal Aspects of Semantic Networks*. Morgan Kaufmann, 1990.

[3] R. J. Brachman and J. G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[4] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer Magazine*, April 1987.

[5] Y. F. Chen and C. V. Ramamoorthy. The c information abstractor. In *Proceedings of the Tenth International Computer Software and Applications Conference (COMPSAC)*, Chicago, October 1986.

[6] T. A. Corbi. Program understanding: A challenge for the 1990's. *IBM Systems Journal*, 28(2), 1989.

[7] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. Lassie: a knowledge-based software information system. In *Proc. 12th International Conference on Software Engineering*, Nice, France, April 1990.

[8] P. Devanbu and D. Litman. Plan-based terminological reasoning. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 128–138, Cambridge, MA, 1991.

[9] Jon Doyle and Ramesh S. Patil. Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence*, 48(3):261–297, April 1991.

[10] S. Feiner and K. McKeown. Generating coordinated multimedia explanations. In *Proceedings of the 6th Conference on Artificial Intelligence Applications*, pages 290–296, Santa Barbara, CA, March 1990.

[11] R. E. Fikes and N. J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[12] G. Fischer and M. Schneider. Knowledge-based communication processes in software engineering. In *Proceedings, Seventh International Software Engineering Conference*, Orlando, FL, 1984.

[13] M. P. Georgeff. Planning. In J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, editors, *Annual Review of Computer Science*, pages 359–400. Annual Reviews Inc, 1987.

[14] W. L. Johnson and Elliot Soloway. Proust: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11(3), March 1981.

[15] S. Kambhampati and J. A. Hendler. Flexible reuse of plans via annotation and verification. In *Proceedings of the 5th IEEE Conference on Applications of AI*, 1989.

[16] H. A. Kautz. *A Formal Theory of Plan Recognition.* PhD thesis, University of Rochester, Rochester, NY, May 1987.

[17] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Aritifical Intelligence*, pages 692–697, Anaheim, CA, 1991.

[18] D. Litman and P. Devanbu. Clasp (classification of scenarios and plans): User's guide. Technical report, Bell Laboratories, 1990.

[19] R. M. MacGregor. A deductive pattern matcher. In *Proceedings of AAAI*, pages 403–408, St. Paul, MN, 1988.

[20] P. F. Patel-Schneider. Small can be beautiful in knowledge representation. In *Procedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, Colorado, December 1984.

[21] Peter F. Patel-Schneider, Bernd Owsnicki-Klewe, Alfred Kobsa, Nicola Guarino, Robert MacGregor, William S. Mark, Deborah L. McGuinness, Bernhard Nebel, Albrecht Schmiedel, and John Yen. Term subsumption languages in knowledge representation. *AI Magazine*, 11(2):16–23, Summer 1990.

[22] P.F. Patel-Schneider and A. Borgida. The (non-standard) treatment of individuals and user functions in classic concept definitions. Technical report, Bell Laboratories, 1991.

[23] C. Rich. A formal representation of plans in the programmer's apprentice. In *Proceedings, 7th International Joint Conference on Artificial Intelligence*, Vancouver, Canada, August 1981.

[24] E. D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier, New York, 1977.

[25] Earl Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[26] R. Sedgewick. *Algorithms.* Addison Wesley, 1988.

[27] P. G. Selfridge. The multiview system. Technical report, Bell Laboratories, 1989.

[28] Candace L. Sidner. Plan parsing for intended response recognition in discourse. *Computational Intelligence*, 1(1):1–10, February 1985.

[29] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5), September 1984.

[30] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for de-localized plans. *Communications of the ACM*, 31(11), November 1988.

[31] J. Steffen. *The CScope Program, Berkeley UNIX Release 3.2*, 1981.

[32] W. Swartout and R. Neches. The shifting terminological space: An impediment to evolvability. In *Proceedings of AAAI*, pages 936–941, 1986.

[33] J. D. Tenenberg. Inheritance in automated planning. In *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, pages 475–485, Toronto, Ontario, Canada, 1989.

[34] M. B. Vilain. The restricted language architecture of a hybrid representation system. In *Proceedings of IJCAI-85*, pages 547–551, Los Angeles, CA, 1985.

[35] W. Wahlster, E. Andre, S. Bandyopadhyay, W. Graf, and T. Rist. Wip: The coordinated generation of multimodal presentations from a common representation. In *Proceedings of International Workshop on Computational Theories of Communication and their Applications: Problems and Prospects*, Trentino, Italy, 1990.

[36] M. P. Wellman. *Formulation of Tradeoffs in Planning Under Uncertainty*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

[37] John Yen, Robert Neches, and Robert MacGregor. Clasp: Integrating term subsumption systems and production systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):25–31, 1991.