# Building Domain Representations from Components[1]

Peter Clark and Bruce Porter
Dept. Computer Science
Univ. Texas at Austin
Austin, TX 78712, USA
{pclark,porter}@cs.utexas.edu

**Abstract**

A major cause of the knowledge-engineering bottleneck is the inability to transfer representational fragments from one knowledge base to another due to the idiosyncratic nature of domain-specific representations. In this paper, we show that representations can be built automatically by composing abstract, reusable components. Moreover, we describe how representations of specific situations, that arise during problem solving, can be assembled 'on demand', guided by a query for a particular piece of information. Our work integrates ideas from dynamic memory, conceptual graph theory, compositional modeling and graph unification.

## 1 Introduction

A major cause of the knowledge-engineering bottleneck [11] is that building one representation contributes little to building the next because each is idiosyncratic. We claim this problem is not inherent to knowledge engineering; rather, it is a limitation of our current technology. Our goal is to develop a new suite of methods suitable for: specifying domain representations as compositions of abstract, reusable components; and assembling these representations, on demand, to answer questions.

Our work is motivated by several observations from our eight year experience building a large-scale knowledge base (KB) in botany [15], and more recently another KB about distributed computing systems:

1. Many domain-specific concepts share similar abstractions, reflected by their representations sharing similar substructures. For example, the general pattern describing `production` recurs in the representation of many concepts in the botany KB, such as photosynthesis, mitosis, growth, and germination. Moreover, many domain-specific concepts appear to be composites of multiple abstractions. Germination, for example, includes conversion, production and expansion.

2. There isn't a single 'right way' of representing a concept: it can be represented in a variety of ways, depending on the intended use of the representation. Again, these variations share similar structure, and there is systemacy in the way substructures are added and removed to form a variant.

3. In our recent work on the distributed computing KB, we achieved very little transfer from the 'top levels' of the botany KB to the computing domain via standard isa-inheritance (henceforth, 'inheritance').

[1]Support for this research is provided by a grant from Digital Equipment Corporation and a contract from the Air Force Office of Scientific Research (F49620-93-1-0239).

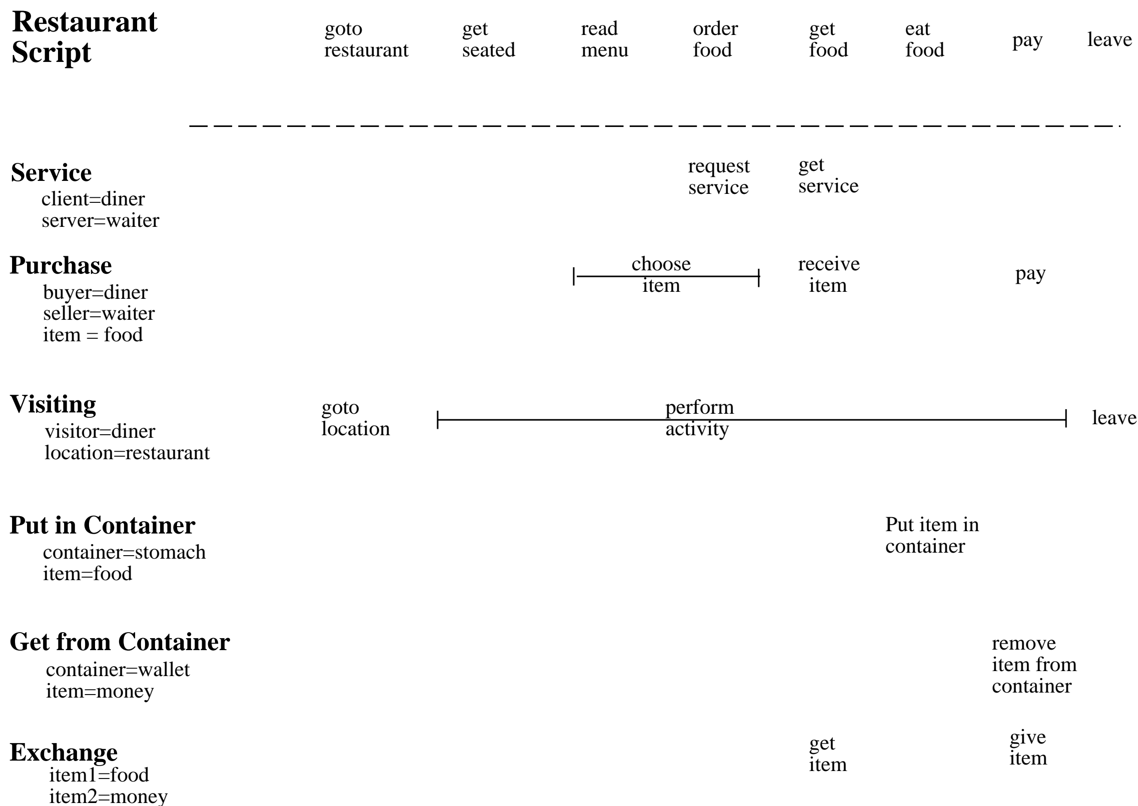



Figure 1: 'The' restaurant script can be viewed as a superposition of more abstract structures.

These observations highlight the importance of developing better methods for representing abstractions and composing concepts from them. As an example of a composite concept, consider the canonical RestaurantVisit script (Figure 1). This stereotypical sequence of events can be viewed as a superposition of various domain-independent abstractions: a purchase, a service, filling a container (the diner's stomach), emptying a container (the diner's wallet), and so on. Informally, it appears the script could be built by connecting together, in some way, these more abstract structures, rather than building a representation from scratch. Moreover, if the representation were an assembly of components, it could be usefully disassembled, to focus on (or ignore) selected aspects of it. Under this view, there is no longer 'the' RestaurantVisit, but rather there are many variants built from various components.

It is also clear from Figure 1 that composition is more than inheritance, as different abstractions overlap and interact. The RestaurantVisit script is not simply a set of abstract scripts, but rather a single script in which the different abstractions have been merged. Although inheritance can collect feature lists from multiple generalizations, it does not offer a generic method for combining structured information. A more sophisticated composition operator is needed.

As Section 2 describes, building representations from components has received widespread attention. Building on this work, Section 3 describes a representation for components, and a method for assembling them, based on a variant of conceptual graphs and graph unification. We attach a semantics to graph structures such that the syntactic operation of graph unification corresponds to the semantic operation of gathering and simplifying multiple constraints, which achieves our goal of integrating information from multiple abstractions.

Having described how representations can be specified as compositions, Section 4 describes how such specifications can be used for reasoning. Following Clancey [7], we view question-answering as constructing a *situation-specific model*, based on initial information and these specifications, about the particular problem being solved. Section 5 shows how this model can be built in a lazy fashion, generating only those parts required to answer questions, rather than attempting to generate the representation in full detail, which is intractable.

All of this work has been implemented (except where noted) and is currently being used to build a knowledge base and question answering system in the domain of distributed computing.

## 2 Prior Work on Components and Composition

Assembling representations from components has received considerable attention in AI, as evidenced by this partial, historical account.

**Frames and Inheritance** Minsky proposed representing information in frame-systems [14]. The components (frames) are organized in a taxonomy, and the composition operator (multiple inheritance) collects the properties of an object, when they are needed, by ascending the hierarchy. However, inheritance does not provide an adequate mechanism for *integrating* this information. For example, if various generalizations provide different values for some relation, inheritance typically returns only the first value, or the set of all of them. What is often needed, however, is some integration of those values into a new, compound value.

This shortcoming of inheritance is well recognized in current software engineering, where software components can inherit methods from multiple abstractions, but cannot combine them. Batory [4] provides an illustration of this in the Booch C++ component library [5]: in which the `guarded_bounded_ordered_deque` and `guarded_unbounded_unordered_queue` classes share only one superclass (`deque`), even though they also use the same concurrency control method (`guarding`). Consequently, the component writer must repeat the code for guarding in both classes. Multiple inheritance would not help because what is needed is an integration, not a concatenation, of the algorithms for deque and guarding. Code repetition is common in libraries [4], and many researchers are studying additional methods for software reuse [3, 20].

**Cliches** Motivated by Minsky's work, Chapman proposed another approach to assembling representations from components [6], which was partially implemented in the Programmer's Apprentice [16]. Chapman described components, called cliches, as "patterns commonly found in representations" — such as Containment, Propagation, and Resistance — from which more specific representations can be assembled. In the Programmer's Apprentice, cliches were represented as parameterized templates, and composition involved instantiating a template's parameters (called roles) with other templates. The composition was quite sophisticated; a template passed as a parameter could be fragmented, and the fragments used in different places in the parent template. Although the design and use of templates was complex, and the composition process had to be carefully guided by the user, the Programmer's Apprentice was a significant advance.

**MOPS** A third approach to assembling representations, proposed by Schank, resulted from his dissatisfaction with the rigidity of scripts [18]. Under this proposal, scripts are assembled, as needed, from more general components called Memory Organization Packets (MOPS). Although several projects devised representations for MOPS, they largely used existing techniques for composing them. CHEF [12], for example, reverted to standard methods such as inheritance, and

BORIS used templates that were manually overlaid on pre-written, detailed scripts [9]. Nevertheless, Schank's proposal for a memory capable of dynamically assembling representations is important inspiration for our work.

**Compositional Modeling** A fourth approach to assembling representations comes from work on compositional modeling, the task of building a model of a physical system adequate for answering questions about the system [10, 13, 17]. In compositional modeling, a component (called a model fragment) contains a set of constraints[2] with well-defined semantics, rather than being represented in syntactic terms as a partial data structure. For example, a component may represent a resistor by a set of constraints relating the resistor's voltage, resistance and current. The composition of a set of model fragments is simply the union of their constraint sets, and reasoning involves constraint satisfaction. We similarly view a component as specifying constraints on a final representation, but expressed as a single data structure rather than a set of statements. This allows much of the work for combining constraints to be performed by a syntactic operation which merges such data structures, rather than using a constraint satisfaction engine.

**Conceptual Graphs** A final approach to assembling representations, which forms the basis of our approach, is the use of conceptual graphs (CGs) [19] and psi-terms [2] to represent components. Both are graph-like data structures containing labeled nodes (concepts) and arcs (relations). Conceptual graph theory provides a variety of logical semantics for different types of graphs. Psi-term theory provides a well-defined syntactic operation called *graph unification* (or *join*, in CG terminology) for merging psi-terms together . Our approach is based on combining these two theories: By attaching an appropriate CG-like semantics to psi-terms, we can ensure that the syntactic graph unification operation is also logically valid, and hence will properly integrate information from components together, as we describe below.

## 3 Components and their Composition

Informally, a component is a description of an object, event, or state, represented as a system of concepts and relations that are packaged together and manipulated as a single unit. Although components might be expressed at any level of abstraction (from concrete instances to class prototypes), we focus on abstract ones, such as `container`, that can contribute to many domain representations. In the spirit of cliches, the `container` component describes an object that:

> partitions a space into two regions, `inside` and `outside`, permitting only two operations, `get` and `put`, to transport objects between these regions, such that the objects pass through the container's `portal`, subject to size and capacity constraints.

Although there are containers that violate this description (eg. sponges containing water, disks containing files), this does not reduce the need for reusable representations of typical containers, it only intensifies the need for methods that adapt representations.

We represent components with 'KM-terms'[3], a variant of conceptual graphs. Figure 2 illustrates their three most common forms. As with conceptual graphs, KM-terms are graphs of concepts and relations (called sorts and features in psi-terms). Concepts are also organized in a taxonomic hierarchy. The root node of a KM-term (eg. `A` in Figure 2b), called its *head*, must be

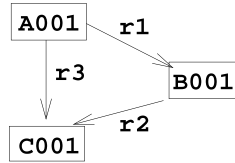
[2] also called a set of 'relations' [13] or 'behavior conditions' [10]

[3] for 'Knowledge Manager', the name of the software managing our KB.

**KM–Term**

```
A001
----
 r1: B001
     ----
      r2: C001
 r3: C001
```

**Graphical Notation**




**Logical Interpretation**

```
r1(A001,B001) & r2(B001,C001)
  & r3(A001,C001)
```

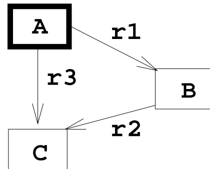
Figure 2a: A KM–term denoting three instances and their relationships. A001, B001, and C001 are anonymous identifiers (Skolem individuals) to indicate the nodes are instances. They are under A,B and C respectively in the concept hierarchy.

---

**KM–Term**

```
A
-
 r1: B
     -
      r2: (the r3 of Self)
 r3: C
```

**Graphical Notation**




**Logical Interpretation**

```
Forall x UniqueExists y UnqiueExists z
  A(x) -> B(y) & C(z) & r1(x,y)
      & r2(y,z) & r3(x,z)
```

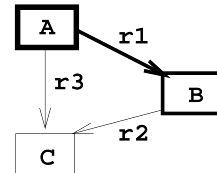
Figure 2b: A KM–term with a concept as its head denotes universal quantification and a logical implication ("All As are ...."). Note the use of a path for coreference from B to C. Note also ‘UniqueExists y’ means ‘there exists exactly one y in relation r1 to x’.

---

**KM–Term**

```
A
-
 r1: B
     -
      definitional?: t
      r2: (the r3 of Self)
 r3: C
```

**Graphical Notation**




**Logical Interpretation**

```
Forall x Forall y UniqueExists z
 A(x) & B(y) & r1(x,y) -> C(z)
     & r2(y,z) & r3(x,z)
```

Figure 2c: A KM–term with the subterm B flagged as definitional, and shown in bold in the graph, denotes a different quantification pattern ("All As in relation r1 to a B are...").

Figure 2: The most common KM-terms and their semantics.

either a concept or a conjunction of concepts — such as {`pet&fish`} — in the taxonomy. *Paths*, such as "`the r3 of Self`" in Figure 2b, express co-reference of sub-terms in a KM-term, where `Self` denotes the head concept and `r3` denotes a relation from it to the shared subterm.

The semantics of KM-terms is based on first-order logic, as illustrated by the examples in Figure 2. Individuals are called instances, and concepts are unary predicates over instances. To denote an instance, we use a unique identifier (Skolem individual), such as `A001` in Figure 2a. To denote a concept (ie. a class), we universally quantify over its members by using its name (rather than a Skolem individual) at the head of the KM-term, as shown in Figure 2b. Other quantification patterns can be expressed by tagging concepts within a KM-term with a special `definitional?` relation (Figure 2c), handled in a special way during inference. We restrict tagging such that tagged concepts are either connected to the head concept or, recursively, to other tagged concepts.

The main challenge for a component-based representation falls on the composition operator: it must be capable of integrating information from various components, not simply collecting it. We implement this capability with *graph unification*, defined in Table 1, based on a similar algorithm for psi-term unification [2]. Our objective with unification is to syntactically merge graphs in a way that corresponds to merging semantically related information. This requires an additional

```
PROCEDURE UNIFY (Term1, Term2)
BEGIN
 1. Find the head concept of the new term by
        (a) unioning the head concepts of Term1 and Term2
        (b) remove any concept which has a subconcept in that union
    (eg. car&vehicle->car, pet&fish->{pet,fish}, {pet,fish}&animal->{pet,fish})
 2. Attach all Term1's and Term2's relations to this head
 3. For all relations which are present in both Term1 and Term2:
        Unify recursively the values (themselves terms) of those relations
END
```

Table 1: The Procedure for Unifying Two KM-Terms.

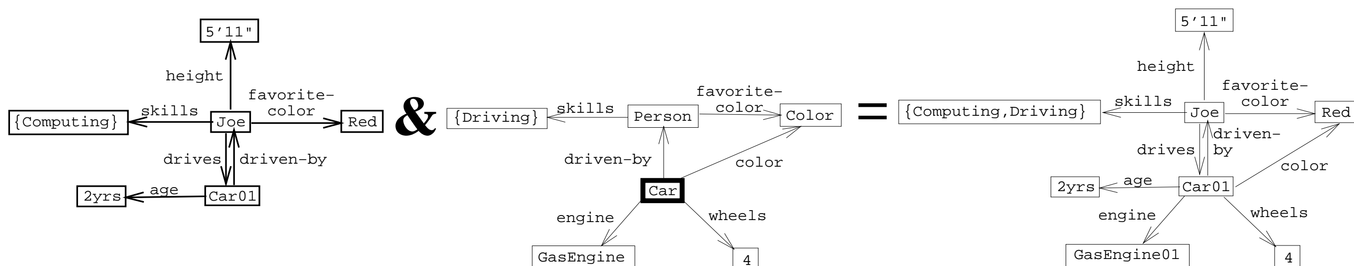



Figure 3: Unification of the KM-term's `Car` and `Car01` (a subterm of the term describing `Joe`). Note how unification integrates information from the two graphs, eg. Joe's skills, Car01's color. (The names `Joe`, `Computing`, `Driving`, and `Red` all denote instances).

constraint on KM-terms: a relation in a KM-term has a *unique value* for its second argument, given its first argument (ie. $\forall xyz R(x,y) \wedge R(x,z) \Rightarrow y=z$). We denote this using the quantifier `UniqueExists`, rather than `Exists`, as in Figure 2b. Under this interpretation, the syntactic operation of unification is semantically valid: nodes can be merged because they refer to the same individual.

Under this semantics, we use partial sets and partial sequences as the fillers for multivalued relations, such as `parts` and `parents`. For example, we write `Pete parts: {head}` rather than `Pete parts: head`, where `{head}` denotes a *partial set*, ie. a unique, but incompletely specified, set containing `head`, and possible other members. Partial sets are unified by unioning their known members — eg. `{a}` and `{b}` unify to `{a,b}` — and then removing members that subsume others — eg. `{a,b}` would become `{a}` if `b` is a superconcept of `a`. This is semantically valid, as the most general, common specialization of a set containing at least `a` and a set containing at least `b` is a set containing at least `a` and `b`. Note we are not unioning two distinct sets, but rather merging constraints on a single set; the unified set has all the constraints (ie. known members) as the initial sets.

KM-terms represent sequences of events in a similar way. The *partial sequence* `<a,b>` denotes a sequence containing `a` and `b` and possibly other events, and where `a` precedes `b`. Two partial sequences unify to a non-linear partial sequence. For example, the unification of `<a,b>` and `<c,d>` is some sequence of steps containing `a`, `b`, `c`, and `d`, in which `a` precedes `b` and `c` precedes `d`. Again, this operation combines constraints. While we should represent the resulting non-linear

sequence using a graph, in practice we restrict ourselves to a linear form[4] by finding the first total ordering consistent with any known constraints (eg. `<a,b,c,d>`).

It is logically valid to unify two KM-terms that have co-referential nodes. One way in which nodes might be co-referential is as follows: the head of one KM-term is a superconcept of an instance in the other (eg. `Car` and `Car01` in Figure 3). In this case, the nodes can be unified (because a superconcept's graph applies to all its instances – see Figure 2b), thereby collecting the information in the KM-terms into one graph.

# 4 Assembling Representations from Components

Given this brief description of components (KM-terms) and composition (unification), we now illustrate how domain representations can be built with components. We continue with the `RestaurantVisit`, as this concept is familiar, is rich in structure, and is known to pose problems for composition [9, 18]. Figure 4a shows a KM-term representing a simplified 'restaurant script', as might typically be encoded in a knowledge base. There is a `diner`, a `waiter`, and a `meal`. During the visit, the `diner selects` his/her `meal`, `requests` it from the `waiter`, `receives` it, `eats` it and finally `pays`.

We can identify numerous abstractions within this structure — `service, purchase, containment, transfer, exchange` — and Figures 4b and 4c illustrate components for the first two. A `Service` is an event in which a `client requests`, then `receives`, some `item` from a `server`. Similarly, a `Purchase` is an event in which a `buyer receives` then `pays` for some `item` from a `seller`. Each component represents a theory about some abstract system of objects. `Purchase`, for example, tells us that the amount paid is the value of the purchase, the seller is the donor, the seller must have the item in order to sell it, and so on. Note that the abstractions are not specific to restaurants, and hence are general in nature.

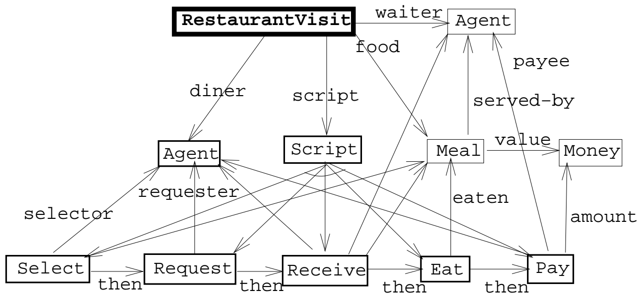
We now have an alternative to building a domain-specific representation from scratch: we can specify which abstractions it is composed of, and unify those components. To do this, we must specify a mapping between terms in the domain and terms in each component, to show how the abstraction applies. For example, in the restaurant visit, a `RestaurantVisit` is a `Purchase` in which the `diner` is the `buyer` and the `waiter` is the `seller`. We can think of each component as having an *interface*, namely the set of terms (eg. `buyer`, `the value of the purchase`) it includes; the mapping connects the interfaces of components when their terms are not identical. Figure 5 illustrates this, by specifying the restaurant visit as a composition of components (just showing two, for simplicity).

Assemblying representations from abstract components simplifies knowledge engineering, and also improves the representation's flexibility:

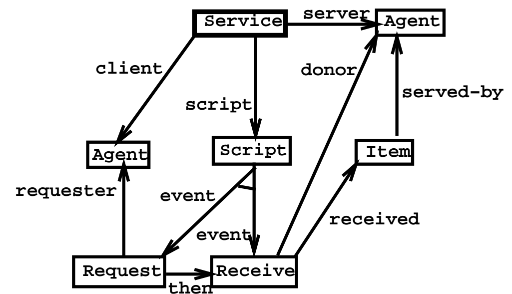
- we can represent atypical visits (eg. carry-out) by varying the components and mappings.
- we can construct the simpliest representation adequate for the current problem-solving situation (eg. determining the price of the meal does not require the `visit` component).
- we can elaborate parts of the representation, as needed (eg. incorporating the `service` component adds information about `servers` performing actions on behalf of `clients`, information that is needed to explain why the waiter delivered the meal to the buyer.)

[4] This is a minor limitation of our implementation.

RestaurantVisit




'Service' component




'Purchase' component

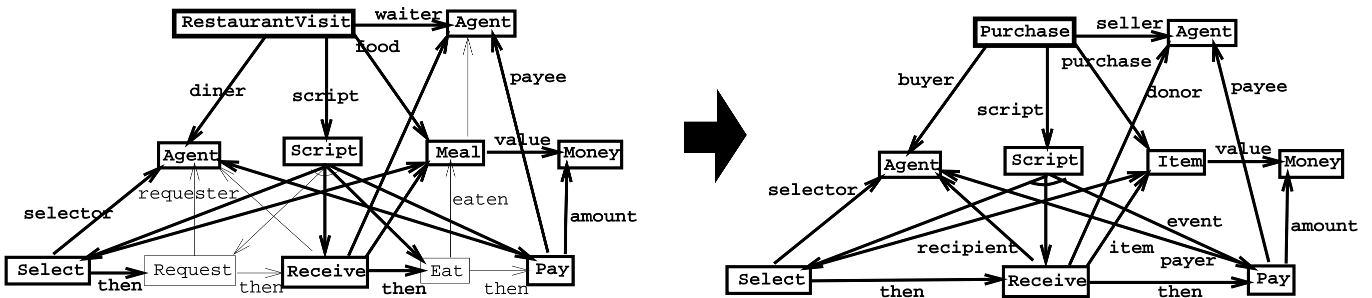



Figure 4: Abstract component structures can be identified within a specific representation of a restaurant visit.

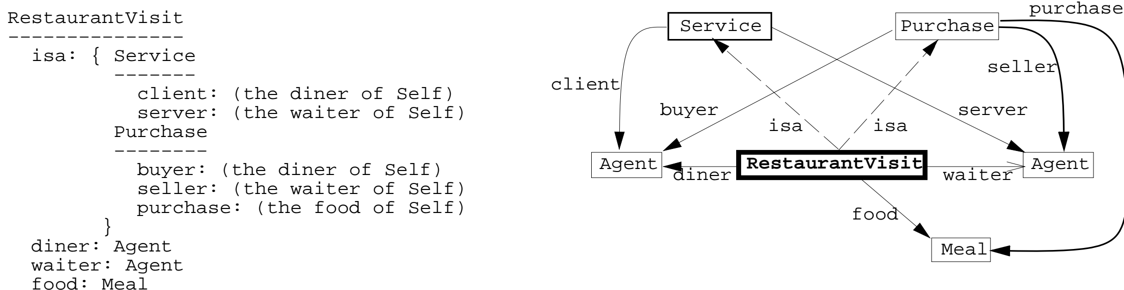



Figure 5: The restaurant visit can be expressed as a composition of abstract components. Note that the mapping among components is expressed as qualifications on the fillers of the *isa* relations.

People routinely exercise this flexibility. For example, in our distributed computing domain, technical manuals describe processes, such as a client-server interaction, in many different ways, depending on context. One description will include security aspects (eg. transmitting authorization credentials), another will ignore security but include addressing aspects (eg. locating the server), and another will ignore both but include transmission protocol information. These variants are analogous to selectively ignoring aspects of the RestaurantVisit (eg. paying, meal production) for a particular restaurant description.

# 5 Building a Representation "On-demand"

We have shown how we can describe domain-specific representational structures by specifying them as unifications of more abstract, general structures. In fact, it would be prohibitive to explicitly build the domain-specific representation in all its detail, as many components can potentially contribute to the representation. For example, in RestaurantVisit, not only do `Service` and `Purchase` components apply but also components describing `Agent`, `Meal`, `Pay`, `Money`, etc. Every node in a graph can potentially 'pull in' another graph, and hence the process is potentially endless. In fact, building the domain-specific representation in all its detail amounts to exhaustive deduction using the knowledge-base.

It is obviously essential to elaborate a representation in a controlled fashion, in response to the demands that a problem-solver (or user) places on it. To exert this control, we use two related mechanisms: path following, where paths through the graphs are used to guide inference, and lazy unification, where we only unify fragments of graphs required to find the answer to a query.

## 5.1 The Query Interpreter

Before describing these mechanisms, we first present the context in which inference occurs using the knowledge base. While the knowledge base describes general concepts such as `RestaurantVisit`, `Service` etc., inference is centered on a specific problem-solving situation (eg. John eating Lobster at a specific restaurant). This initial set of facts is represented by a graph of instances that we call the **instance graph**. The instance graph is elaborated by unifying it with components from the knowledge base, each unification constituting a single step of inference.

A **query interpreter** mediates between a problem-solving algorithm (or the user) and the knowledge base. Inference is triggered when the interpreter receives a query. A query asks

for information from the instance graph (or the deductive closure of it), and is expressed by a path (Section 3), eg. `the server of the meal of RestaurantVisit01`. The interpreter then performs the necessary inferences to answer the query (we say it **evalutes** the query) using the mechanisms described below.

## 5.2 Path Following

One mechanism the query interpreter uses for guiding search while evaluating a query is *path following*. A path, eg, `the server of the meal of RestaurantVisit01`, does not just refer to a node in the instance graph: it also expresses additional information about *how* the value of that node can be computed, namely, first find the meal of the restaurant visit, then find its server. The query interpreter follows paths in this way, using them as specifications of a sequence of inference steps (here, unifications) that result in the desired value. Without this, there is a potentially large search required to identify a chain of inference which will find the answer.

Similarly, the value found at the end of a path may itself be a path (recall that paths are used to express co-reference within KM-terms (Figures 2b and 2c)). For example, consider the KM-term describing `Purchasing`:

```
Purchasing
----------
 script: Script
         ------
          events: <... Pay >
                       ---
                       payer: (the buyer of Self)
                       amount: (the cost of the purchase of Self)
```

If the query interpreter is asked for the amount of the pay event (of some instance of Purchasing), it will find (from this KM-term) a path for this value, namely `the cost of the purchase of Self` (where `Self` refers to that instance). The query interpreter evaluates this new path (using loop-checking to avoid cycles), until it finds a non-path value or fails. This illustrates how path-following can trigger further path-following. It also illustrates that paths within KM-terms have a second function besides expressing co-reference: they provide guidance to the query interpreter about *how* a value can be computed, by specifying a sequence of inference steps leading to that value.

Of course, this efficiency trades off completeness – there could be other sequences of inferences that reach the conclusion, but are not explored because they are "off the path". However, we retain a useful, albeit weakened, type of completeness called Socratic completeness [8]. Socratic completeness guarantees that any deductive consequence of the KB is deducible via some sequence of queries. Thus, no logically implied fact is inherently indeducable, and hence the knowledge base designer has full control over where and where not inference effort should be expended by his/her choice of paths in the knowledge base.

## 5.3 Lazy Unification

A second, and related, mechanism the query interpreter uses to control inference is *lazy unification*. Unifying two KM-terms involves merging their graph structures. However, when evaluating a path, the query interpreter only needs the result of unifying the *branch* of the graph specified by the path with its counterpart in the other graph. In this way, lazy unification avoids the unnecessary work of unifying the entire graphs.

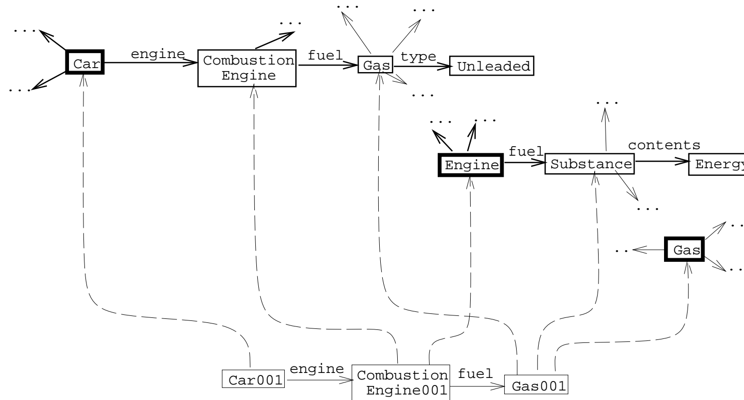



Figure 6: Lazy unification involves just unifying graphs along a particular branch. Pointers are maintained in case a subsequent query requires the partial unification to be completed further.

However, note that lazy unification requires one extra step. Answering subsequent queries may require unifying other portions of the KM-term's that were skipped due to laziness. To prepare for this eventuality, the query interpreter installs pointers from nodes in the unified graph to the components (KM-terms) from which they came. This gives the query interpreter a handle on the partially unified KM-terms in case the unification must proceed further.

This is illustrated in Figure 6, in which graphs for `Car`, `Engine` and `Gas` have been partially unified along the path `the fuel of the engine of a Car`. Note that the pointers from `Gas001` to the structures `Car` and `Engine` provide handles on the information those components can still contribute, such as the `type` or `contents` of `Gas001`. This information would be explicitly added to the instance graph should a subsequent query request this information. Note too that `the type of Gas001` could not be concluded by normal inheritance (via `Gas001 isa Gas`) as the answer resides on the `Car` component, illustrating unification's ability to integrate information from multiple sources (here `Car`, `Engine` and `Gas`).

## 5.4 Extended Example from the RestaurantVisit

We now present a more detailed example of answering a query to illustrate path following and lazy unification. The example is intended to show a real-world operation, namely scanning a sequence of events for possible failure points, but is couched in terms of our ongoing restaurant example. In the tutoring system for distributed computing that we are building, this operation is frequently used in the diagnostic component to identify failure points in distributed computing scenarios, in order to explain an error the user has observed.

In the context of the restaurant visit, a diagnostic problem-solver might ask many questions to identify potential failures, such as: "What are the preconditions for the events in the script?" Expressed as a path, this query would be:

```
[1]-> the preconditions of the events of the script of a RestaurantVisit
        with diner = John and food = a Lobster?
```

This query is first broken down to evaluate its innermost expression, namely the description of this particular restaurant visit:

```
[2]-> the events of the script of a RestaurantVisit with ...?
```

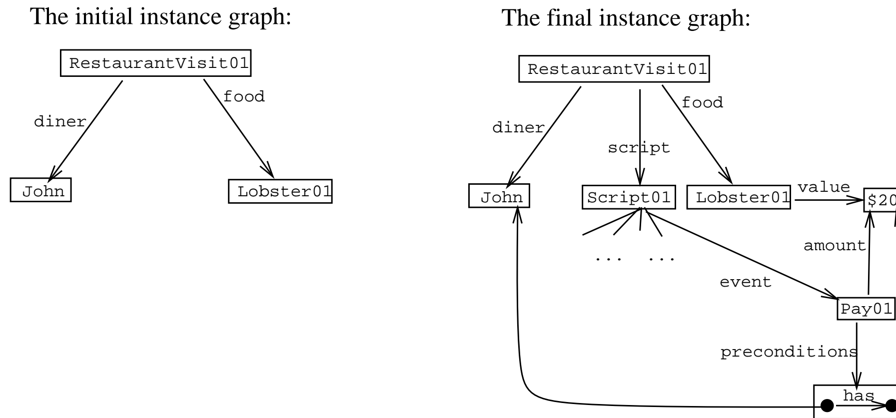



Figure 7: An initial graph is grown only as needed to answer a query (here for the events' preconditions)

```
  [3]-> the script of a RestaurantVisit with ...?
   [4]-> a RestaurantVisit with diner = John and food = a Lobster?
```

This expression tells the query interpreter to create an initial instance graph representing this restaurant visit, as illustrated in Figure 7.

```
   [4]<- RestaurantVisit01                         ;; return the new instance
```

Now the interpreter can re-evaluate step `[3]`, substituting in the value of the restaurant visit:

```
  [3]-> the script of RestaurantVisit01?
```

The instance graph currently has no `script` arc from `RestaurantVisit01`, causing the interpreter to incorporate, with lazy unification, those components of `RestaurantVisit` that contain a `script` arc. Following Figure 4, two are found, namely `Service` and `Purchase`. A new instance `Script01` is created, denoting the script of `RestaurantVisit01`, and pointers are added from it to the `Script` nodes in `Service` and `Purchase`.

```
  [3]<- Script01                                   ;; from Service and Purchase
 [2]-> the events of Script01?
```

Similarly, the interpreter incorporates components providing events for this script. Both `Service` and `Purchase` provide sequences, which are interleaved, as described in Section 3, to return a single sequence. Again, instances are created and stored in the instance graph.

```
 [2]<- <Select01 ... Pay01>                        ;; from Service and Purchase
[1]-> the preconditions of <Select01 ... Pay01>?
```

For brevity we show only the `Pay01` branch:

```
[1]-> the preconditions of Pay01?
```

Again, `Pay01` has no `precondition` relations in the instance graph, so the interpreter finds components that can contribute that information. `Purchase` does not specify `preconditions`, but `Pay` itself is specified as including a `Give` component, in which the `giver` (`payer`, as instantiated in the `Pay` component) must have the `given` (mapped to the `amount` in the `Pay` component).

```
[1]<- (payer of Pay01) has (amount of Pay01)       ;; from Pay
```

This returned value is itself a structure consisting of two paths and a relation (`has`). Each path must be evaluated.

```
[2]-> payer of Pay01?
[2]<- the buyer of RestaurantVisit01            ;; from Purchase
[2]-> the buyer of RestaurantVisit01?
[2]<- the diner of RestaurantVisit01            ;; from RestaurantVisit
[2]-> the diner of RestaurantVisit01?
[2]<- John                                      ;; from the instance graph
```

From `Purchase` (Figure 2c), the `payer` in the `Pay` is the `buyer` in the `Purchase`. Then from `RestaurantVisit` (Figure 5), the `buyer` in this `Purchase` is mapped to the `diner` in the `Restaurant`. Similarly for the `amount of Pay01`:

```
[2]-> amount of Pay01?
[2]<- the value of the meal of RestaurantVisit01  ;; from Purchase
[2]-> the value of the meal of RestaurantVisit01?
 [3]-> the meal of RestaurantVisit01?
 [3]<- Lobster01                                  ;; from the instance graph
[2]-> the value of Lobster01?
[2]<- $20                                         ;; from Lobster
```

Again, a simple graph describing `Lobster` (including its typical price) supplies this information.

```
[1]<- John has $20
```

Finally, this returns (one of) the script's preconditions, that John has $20, which identifies a potential failure mode of this script, that John lacks $20. The final instance graph is illustrated in Figure 7.

There are several important points to note from this example. First, computing the answer is not trivial: preconditions for restaurant events are not specified on the `RestaurantVisit` component, but instead are distributed among other, more abstract, components (eg. `Give`) comprising the restaurant visit's specification. Second, the answer is *situation-specific*; if the meal had not been lobster then the required amount would have been different. Third, the instance graph has been elaborated along only those arcs required to answer the query, illustrating 'lazy unification'.

### 5.5 Application of the Representational Framework

We are currently applying this representational framework in two ways. First, we are constructing an automated assistant for users of distributed computing systems, capable of answering novice users' questions. The assistant contains three simple problem-solving algorithms. The first generates definitions of computing terminology by assembling component-based representations of domain concepts and converting them to text. The second performs diagnosis by constructing computing scripts and scanning them for failure points that explain the user's observations (iterating between data-gathering from the user and reasoning with information in the KB). The third generates short plans for achieving a user's goals (eg. "how do I reduce the minimum allowed password length") with a standard means-ends analysis algorithm, using planning operators built compositionally from the knowledge base. All three systems reason about a user's specific computing situation, represented as an instance graph in the knowledge base. This includes representation

of the user's particular computing environment (eg. machines, their connectivity, processes running on them), and any particular activity which is being reasoned about. For example, if the user is asking about possible failures in a binding event (say), then an instance of binding event is created in the instance graph. The application systems query the knowledge base for particular pieces of information about the user's situation, and the query interpreter answers those queries by (lazily) unifying components with the instance graph.

Second, and more importantly, we are in the early stages of constructing a small library containing reusable components such as `communication`, `containment`, `exchange`, and `information`. While the library's contents are primarily being used as building blocks for the computing knowledge base (eg. a `database` is composed of `container`, `secure-item` and `resource`), our goal is to formalize the components in domain-general ways.

# 6 Summary

A major cause of the knowledge engineering bottleneck is the difficulty of building domain representations from more abstract components. Although many domain-specific concepts are composites of many abstractions, it is difficult to represent such abstractions and automatically compose them together.

To address this problem, we have presented a novel representation of components, evolved from a combination of conceptual graph theory and psi-term unification. We have illustrated how graph unification, used as a composition operator, can properly integrate components into a single structure. This approach offers a way to build domain-specific representations from reusable components.

Finally, we have described how situation-specific representations can be assembled from components on demand, guided by a query for a particular piece of information. The query interpretation algorithm utilizes two novel techniques – path following and lazy unification – to guide and constrain inference to just that required to answer a query.

# References


[1] H. Ait-Kaci and A. Podelski. Towards a meaning of LIFE. *Logic Programming*, 16:195–234, 1993. (also available as http://www.isg.sfu.ca/ftp/pub/hak/prl/PRL-RR-11.ps.Z).

[2] H. Ait-Kaci, A. Podelski, and S. C. Goldstein. Order-sorted feature theory unification. Tech Report PRL-RR-32, Digital Paris Research Labs, May 1993. (http://www.isg.sfu.ca/ftp/pub/hak/prl/PRL-RR-32.ps.Z).

[3] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Oct 1992.

[4] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings ACM SIGSOFT'93 (Symposium on the Foundations of Software Engineering)*, Dec 1993.

[5] G. Booch and M. Vilot. The design of the C++ booch components. In *OOPSLA'90*, pages 1–11, Oct 1990.

[6] D. Chapman. Cognitive cliches. AI Working Paper 286, MIT, MA, Apr 1986.

[7] W. J. Clancey. Model construction operators. *AI*, 53(1):1–116, 1992.

[8] J. M. Crawford and B. J. Kuipers. Algernon – a tractable system for knowledge-representation. *SIGART Bulletin*, 2(3):35–44, June 1991.

[9] M. G. Dyer. $RESTAURANT revisited, or 'lunch with boris'. In *IJCAI-81*, pages 234–236, 1981.

[10] B. Falkenhainer and K. Forbus. Compositional modelling: Finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.

[11] E. A. Feigenbaum. The art of artificial intelligence: Themes and case studies of knowledge engineering. In *IJCAI-77*, pages 1014–1029, Cambridge, MA, 1977.

[12] K. Hammond. CHEF: A model of case-based planning. In *AAAI-86*, 1986.

[13] A. Y. Levy. Irrelevance reasoning in knowledge-based systems. Tech report STAN-CS-93-1482 (also KSL-93-58), Dept CS, Stanford Univ., CA, July 1993. (Chapter 6).

[14] M. Minsky. A framework for representing knowledge. In R. Brachman and H.Levesque, editors, *Readings in Knowledge Representation*. Kaufmann, California, 1985. (originally in The psychology of computer vision, Ed: P.Winston, NewYork: McGraw-Hill 1975).

[15] B. W. Porter, J. Lester, K. Murray, K. Pittman, A. Souther, L. Acker, and T. Jones. AI research in the context of a multifunctional knowledge base: The botany knowledge base project. Tech Report AI-88-88, Dept CS, Univ Texas at Austin, Sept 1988.

[16] C. Rich and R. C. Waters. *The Programmer's Apprentice*. ACM/Addison-Wesley, Reading, MA, 1990.

[17] J. W. Rickel. *Automated Modeling of Complex Systems to Answer Prediction Questions*. PhD thesis, Dept CS, Univ. Texas at Austin, 1995. (Also available as Tech Rept AI-95-234).

[18] R. Schank. *Dynamic Memory*. Cambridge Univ. Press, 1982.

[19] J. F. Sowa. *Conceptual structures: Information processing in mind and machine*. Addison Wesley, 1984.

[20] H. Ossher and W. Harrison. Combination of inheritance hierarchies. In *OOPSLA'92*, pages 25–40. ACM, Oct 1992.