# Constructing Scripts from Components: Working Note 6

Peter Clark and Bruce Porter
Dept. CS, UT Austin
{pclark,porter}@cs.utexas.edu

## 1 Introduction

### 1.1 Scripts and Composition

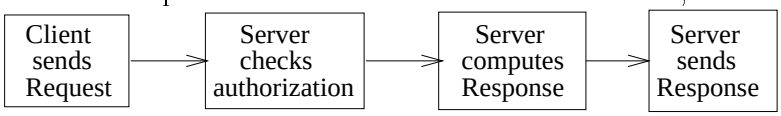
In the earlier working notes, we presented how *object descriptions* (eg. "kayak") and *action rules* (eg. "get") could be composed from components. This working note discusses the third, and last, class of representational structure in our KB, namely *scripts*.

In its simplest form, a script is simply a sequence of actions. Scripts can be used to describe stereotyped action sequences which occur in the world, and can be employed for a variety of tasks including explanation, diagnosis, and prediction. A script is like a plan, except that the rationale for its action sequence might not be fully represented. Script-like structures have been extensively used in AI, in particular for natural language understanding [Schank and Abelson, 1977, DeJong, 1979, Cullingford, 1978].

Despite their utility, treating scripts as a self-contained chunks of knowledge is problematic for several reasons. First, there are many alternative ways of describing some real-world activity (eg. visiting a restaurant); what should be in "the" script, then, and what should not? Second, abstractions common to different scripts are not captured or shared. Ideally, we would like to be able to identify and extract abstractions in a script for explanation or problem-solving purposes. Finally, each new script has to be written from scratch, often repeating similar structures from previous scripts. Instead, we would like to construct scripts by composing more abstract patterns together.

These limitations of scripts as self-contained chunks are well known, and more compositional approaches have often been advocated in the literature. Schank's MOPs (Memory Organization Packets) were originally conceived as more general chunks which could be combined together [Schank, 1982], and Dyer's BORIS system allowed abstractions to be (manually) overlayed on a base script, as depicted in Figure 1 [Dyer, 1981]. The recurring notion of *cliches* is based around the theme of composing intermediate-level descriptions to represent some specific, possibly time-varying, phenomenon [Chapman, 1986, Rich and Waters, 1990].

Our work in concept composition was originally motivated by similar concerns. Consider trying to write a script for "client-server interaction" in DCE, for example:



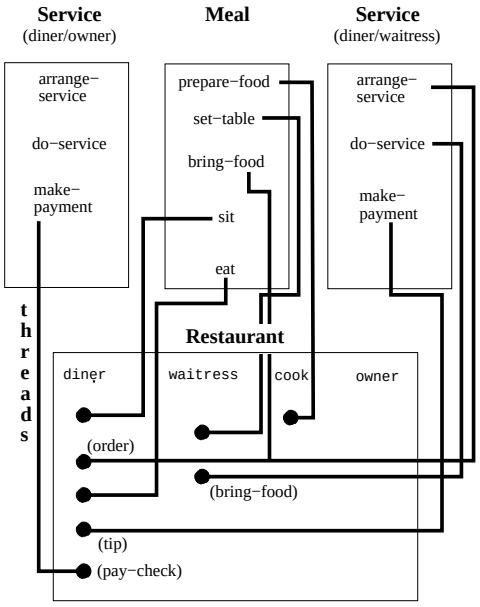



Figure 1: A script can be viewed as a composition of abstractions (from [Dyer,1981]).

We are immediately faced with the issues of what should go in the script? and where are its boundaries? The above 4-step script is not always adequate for problem-solving: for some tasks we may not want to include `authorization` activities, whereas for others, we may be *only* interested in authorization aspects. And what about other actions which we've ignored eg. the client locating the server? Instead, it seems preferable to consider the client-server script as a composition of different threads or components, as informally sketched in Figure 2. This would provide us with a mechanism to remove or include different aspects of the script as required for the task at hand. In addition, we may be able to pattern-match other components into the script which we hadn't originally considered, to provide new descriptions of the activity. Finally, by partial-matching a scriptal component with an incomplete script (eg. provided by the user) we can form expectations about what might occur next, and preferences about how new information should be interpreted. We discuss these possibilities in Section 3 of this paper.

Scripts bear many similarities to computer software – both are sequences of actions which perform some task. Our work on building scripts from components thus takes us close to work on software composition and reuse. In our model below we draw heavily from the inspiring work by Don Batory's software group at Univ. Texas at Austin [Batory, 1995].

## 1.2 Overview

To give a flavor of the script composition method described here, have a quick look ahead to Figure 8. This figure illustrates how a particular script representing "a visit to a restaurant" is constructed from more general components (of types "acquisition", "client-server interaction", "fill", and "exchange"). In fact, the large box doing the combining is itself a component (of type "restaurant-visit").

Section 2 describes components and the composition machinery in more detail. In particular, it describes how the components (eg. "acquisition") are appropriately in-

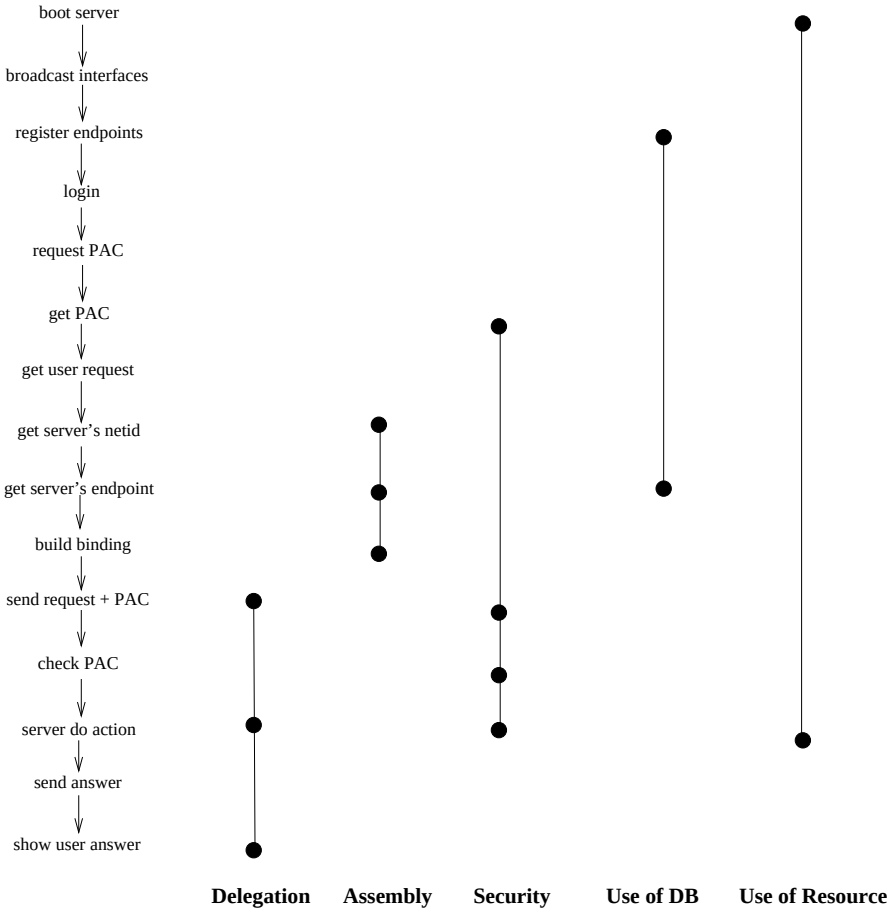



Figure 2: Some components within a script describing DCE interaction.

stantiated through parameter passing, and how component interchangability is achieved through the definition of standardized interfaces, or *realms*.

Section 3 describes a variety of ways in which composed scripts can be usefully employed for problem-solving, not possible without a compositional approach.

Finally Section 4 compares our compositional approach to the work of Batory's group, highlighting where we have adopted and where we have departed from their approach.

# 2 Components

## 2.1 Introduction

### Constructing Scripts

A *scriptal component* (henceforth just 'component') is the basic building block for constructing scripts. A component connects with sub-components, receives fragments of a script from those sub-components, combines those fragments into a script, and outputs that script. A fragment is either an action rule[1] or itself a script (possibly composed from sub-sub-components).

[1]Briefly: an action rule is some data structure describing that action (eg. using preconditions, add and delete lists). Action rules are discussed in more detail in earlier Working Note 4.

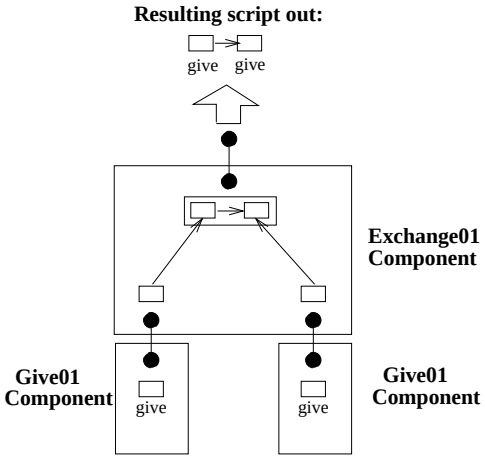



Figure 3: The `exchange01` component joins two `give` actions together to form a script.

---

In the simplest case, the fragment delivered by each sub-component will simply be an action rule, and the component will assemble the action rules into a sequence. Figure 3 illustrates this simple case, where the sub-components supply two `give` action rules ("A gives to B", "B gives to A") to the `exchange01` component, and `exchange01` assembles these into a script ("A gives to B then B gives to A"). Here, the component has simply put these two actions into an order.

### Passing Parameters

As well as combining fragments, a component performs a second important task of instantiating those fragments correctly. To do this, it maps its own 'input parameters' onto the parameters of its sub-components appropriately.

For example, the `exchange` component takes as input two agents (`agent1` and `agent2`) and two objects which are exchanged (`obj1` and `obj2`). In the first `give` action in the script, we wish to express that `agent1` gives `obj1` to `agent2`. To do this we map the `exchange` parameters onto the first `give` component's parameters as follows:

| `exchange` **parameters** | map to | `give` **parameters** |
|---|---|---|
| `agent1` | $\leftrightarrow$ | `donor` |
| `agent2` | $\leftrightarrow$ | `recipient` |
| `obj1` | $\leftrightarrow$ | `object` |

The full mapping for both components is give in Figure 4.

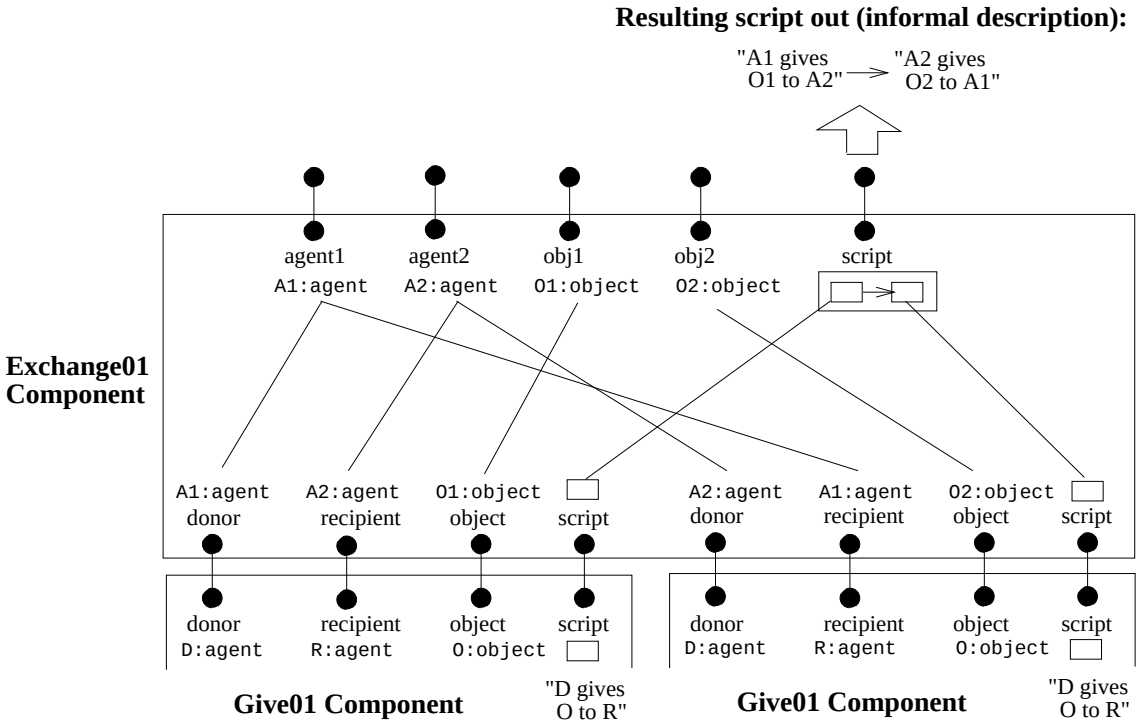



Figure 4: Mapping of parameters in the `exchange01` component.

## 2.2 The Anatomy of a Component

We now describe a component in more general terms. Figure 5 illustrates a component's general structure.

The *interface* to a component consists of a set of *parameters* and a set of *blocks*. We refer to the parameters and the blocks in an interface as its *terminals*. In our examples here a block is a script, but more generally a block can be an arbitrary structure (eg. it might be some program code). Similarly, we only consider interfaces with one block but in general an interface can have an arbitrary number of blocks. It's useful to think of the parameters as the input to the component, and the block as the output. However, as

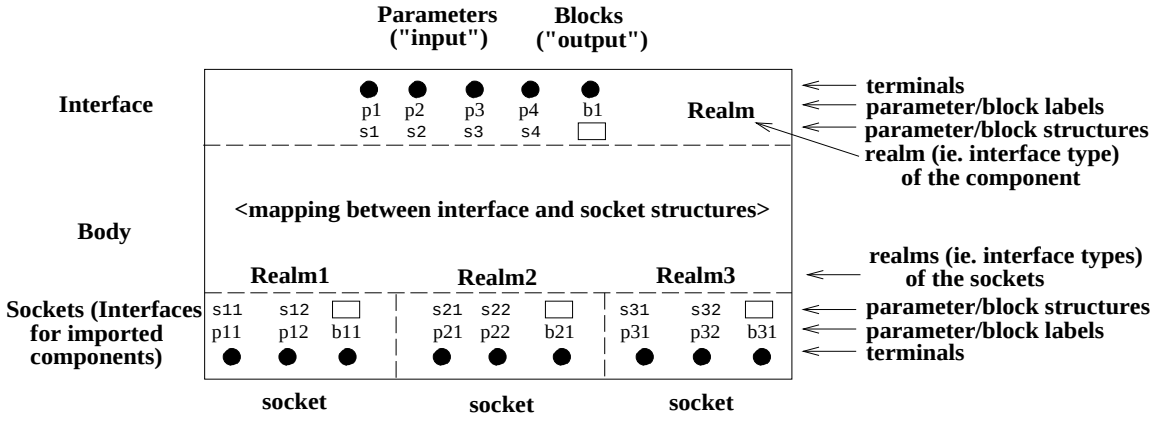



Figure 5: The anatomy of a component.

we shall see later, for the classes of components considered here we can also use them 'in reverse' (ie. the block is the input and the parameters are the output). This reversibility has some useful applications which we describe in Section 3.5.

Each terminal of an interface has a *label* to identify it, and also a *structure*, describing the form of the data which passes through that terminal. The structure may simply be a data type, or a more complex assemblage: for example, the **script** terminal of the `Exchange01` component in Figure 4 has a more complex structure specified. Structures allow us to access the internals of the data passing through a terminal – ie. the data flowing through is *transparent* to the component. In our application, all data and structures are conceptual graphs, and unification is used to match data to structures.

A component also has zero or more *sockets*. A socket is a component's interface to one of its sub-components. They are the ports where the sub-components dock.

The component's body defines a *transformation* between its interface and its sockets. In most cases we simply pass interface parameters to the appropriate socket parameters unchanged (eg. Figure 4), but in general we can also transform or combine the parameters (eg. some socket parameter = 2 $\times$ some interface parameter). Similarly, the interface block and socket blocks are mapped together via some transformation. In our case, the blocks are scripts and the transformation is to combine the socket scripts into a larger script. More generally the transformation could be arbitrary (eg. blocks could be program code, and the transformation could involve wrapping additional code around the socket blocks).

## 2.3 Realms

A component's interface also has a type, or *realm*. A realm is defined by a name (eg. `Give`) and a set of terminal labels (eg. `donor`, `recipient`, `object`, and `script`). All components with the same realm model the same fundamental abstraction (eg. "giving something"), but do so in different ways. We sometimes will write that a component "is a member of a realm R" meaning its interface is of realm R.

Realms are a fundamental concept for composition, as they define *standardized interfaces* for components. In other words, they ensure (in our case) that alternative representations of the same abstract concept will have the same interface, thus allowing us to "plug-and-play" when constructing representations.

As a (simplified) example of this "plug-and-play", consider the representation of an `exchange` shown earlier in Figure 3. The `Exchange01` component imported two components of realm `Give` (namely `Give01` and `Give01`). However, other `Give` components could have been used instead, to construct alternative representations of an `exchange`. As they share the same realm, they are guaranteed to 'fit'. Figure 6 illustrates an alternative composition, where components `Give02` and `Give01` have been connected with `Exchange01`, producing a different script as output.

## 2.4 Type Equations

As in the GenVoca method of software reuse (discussed later in Section 4), we can express a composition of components using a *type equation*. In a type equation, components are considered to be parameterized by (the sub-components filling) their sockets, and the type equation then describes the nesting (connections) of components to form a complete representation. For example:

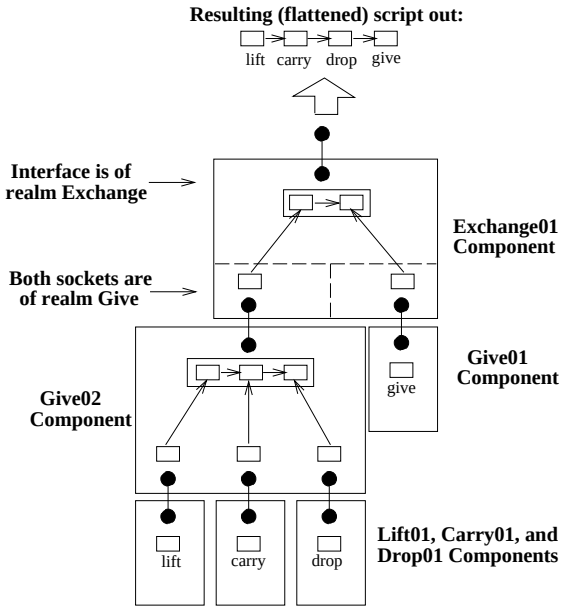



Figure 6: An alternative representation of an exchange, using different components.

---

```
THE COMPONENT LIBRARY
  Components in Realm Exchange = { Exchange01[Give,Give], Exchange02 }
  Components in Realm Give     = { Give01, Give02[Lift,Carry,Drop], Give03 }
  Components in Realm Lift     = { Lift01 }
  Components in Realm Carry    = { Carry01 }
  Components in Realm Drop     = { Drop01 }

SOME COMPOSITIONS
MyExchange01 = Exchange01[ Give01, Give01]                                  (Figure 3)
MyExchange02 = Exchange01[ Give02[ Lift01, Carry01, Drop01], Give01]]]  (Figure 6)
```

The compositions are, of course, themselves components and could be added to the component library (hopefully with more appropriate names).

## 2.5 Plug-Compatibility

Components of the same realm are, in the simplest case, fully interchangable. However, there is an additional constraint on whether a component can connect with a socket (we say the component and socket are *plug-compatible*), namely the *structures* of data at corresponding terminals must match. Loosely speaking, the data the socket provides must match the data the component expects, and vice versa. For example, a `Give` component whose `donor` terminal has a structure `person` (ie. expects the donor to be of type `person`) will not connect with a `Give` socket whose `donor` terminal exports an object of type `database-server`.

This allows us to create special sub-classes of some abstraction. For example, in the realm `Exchange`, we can create components describing an exchange between people, an exchange between computer programs, an exchange between a person and a computer,

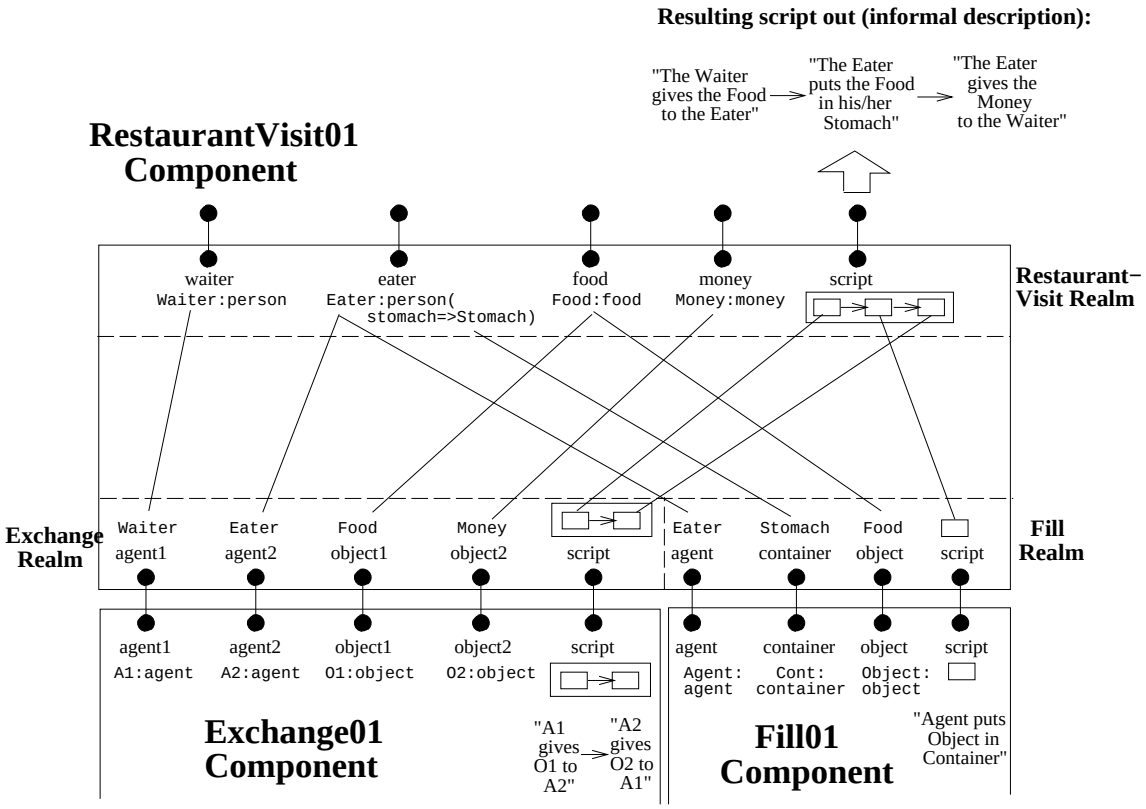



Figure 7: The (simplified) Restaurant Script, composed of components.

etc. Each component will restrict the structures expected/provided at its terminals appropriately. A socket which restricts the structures at its terminals will then only be plug-compatible with those components which have comparable restrictions (ie. the terminals' structures will unify).

## 2.6 Another Example

A slightly more sophisticated example is shown in Figure 7. This shows a (rather simple) description of visiting a restaurant, formed from composing scripts describing an "exchange" (food for money) and a "fill" (of one's stomach).

There are two important things to note in this example. First, the structure for the `eater` terminal is more than just a datatype – instead, it is a (conceptual graph representing a) person and his/her stomach. This allows us to access the internals of the data which this component receives, hence passing `Eater` and `Stomach` as the `agent` and `container` of `Fill01` respectively.

Second, the `RestaurantVisit01` component expects the script passed up from `Exchange01` to have the structure of two consecutive `Give` blocks (this deviates from our previous examples where the script could have any structure). Thus, only `Exchange` components which satisfy this requirement are plug-compatible with this socket. We require access to the structure of the `Exchange` script as we wish to interleave the `Fill` block between the two `Give` blocks. (Note that further sub-structure is still unrestricted though, ie. those

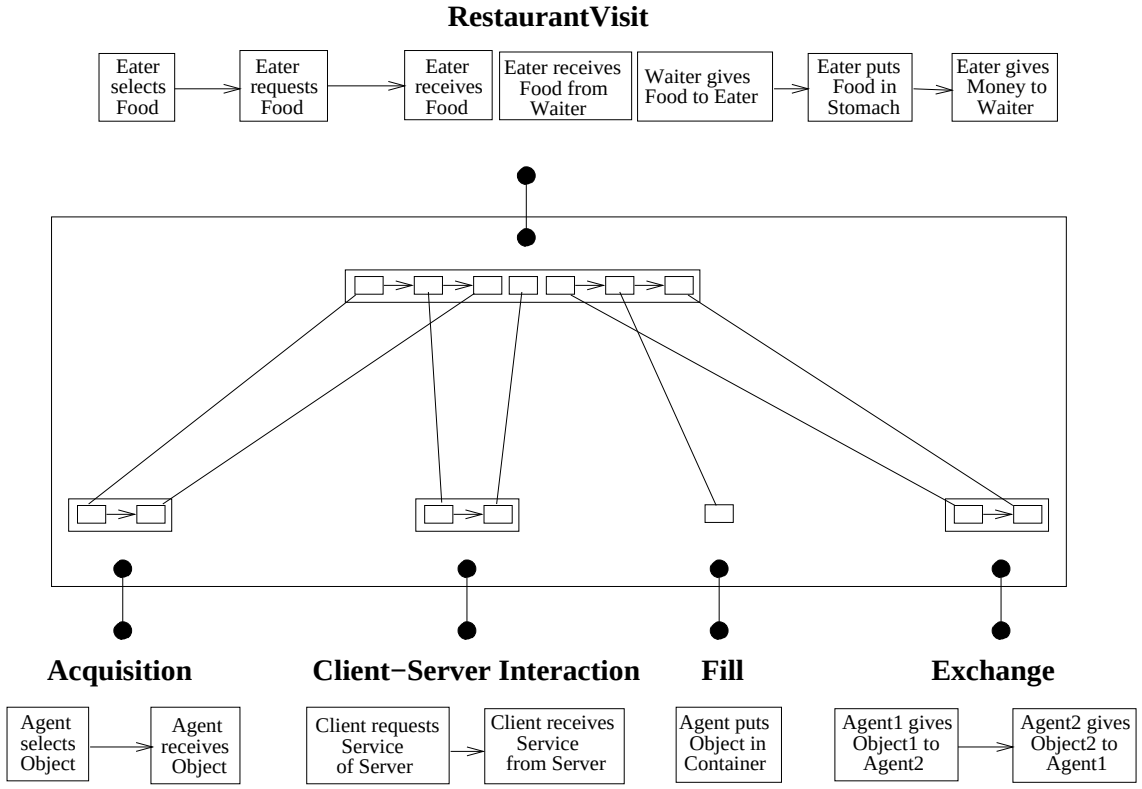



Figure 8: Composition of a `RestaurantVisit` Script from components, including coreferential actions.

`Give` blocks could themselves be actions, scripts, nested scripts, etc.).

## 2.7 Co-referential Actions

It may be that action rules in different sub-components are *co-referential*, ie. refer to the same event. The action rules may be identical, or they may describe different aspects of same action. In the diagrams in this paper, coreferential actions are denoted by no arrow connecting them. Implementationally, the script data structure tags the actions as being coreferential. Figure 8 gives an example which includes coreferential actions. The example shows a `RestaurantVisit` script being composed from components (we haven't shown the parameter passing, to avoid cluttering the diagram). In this example, the receipt-of-food step in the final script has three boxes describing "Eater receives Food", "Eater receives Food from Waiter", and "Waiter gives Food to Eater", which arose from three different sub-components. These three boxes depict three different ways of describing the same action, and constitute different *views* of that action. Different views may describe the preconditions/effects of an action in different ways, and may generate different natural language text. We elaborate on how multiple descriptions of the same event can produce different *viewpoints* in Section 3.4.

As a side point of interest, note that, as in Figure 7, the `RestaurantVisit` component accesses the internals of the sub-components' scripts ('breaks them open') in order to interleave (rather than append) them together in the final script in Figure 8.

# 3 Reasoning with Composed Scripts

## 3.1 Simulation, Goal Analysis and Diagnosis

A sript is a sequence of action rules, each describing how the world changes as a result of that action being performed. In our representation, action rules consist of preconditions, add and delete lists. This allows us to perform a variety of tasks, including:

**Simulation:** Given a starting state, we can 'execute' the script by applying each action rule in turn, hence simulating the changes in the world which occur.

**Goal Analysis:** Given a precondition/add/delete list representation of actions, we can perform goal analysis to answer questions about *why* a particular action is performed. This involves identifying which future actions were 'enabled' (had their preconditions satisfied) by the effects of the action being queried about.

**Diagnosis:** By inspecting the preconditions list of actions, we can perform simple diagnosis by identifying possible failure modes in the script (namely preconditions which might not be satisfied at a given step), and then filtering out those which are inconsistent with symptoms the user observed.

These tasks can, of course, be performed without using a compositional approach to construct the script. Rather, the role of compositionality is to provide versatility in the input to these reasoning mechanisms.

## 3.2 Generating Natural Language Text

As is perhaps apparent from the previous diagrams (observe the quoted text strings in them), it is a simple extension to generate natural language text from the scripts by attaching text templates to actions in scripts. A text template is a 'parameterized sentence', in which some words/phrases in a sentence are replaced with variables to denote what they should refer to. For example in Figure 7 the `Exchange01` component could have the text template:

```
[A1,"gives",O1,"to",A2]
```

attached to its first action. Unification of this component's interface with the `RestaurantVisit01` socket causes these variables to be bound, and so in the final script, output by `RestaurantVisit01`, it will have the form:

```
[Waiter:person,"gives",Food:food,"to",Eater:person]
```

where `Waiter`, `Food` and `Eater` are parameters of `RestaurantVisit01`. (`Waiter:person` denotes that the variable `Waiter` is constrained to be of type `person`). By instantiating the parameters of the script, the script is instantiated as a result. For example, if we set the parameters of `RestaurantVisit01` to be:

```
waiter: "Jeeves"
eater:  "Albert"
food:   "the hamburger"
money:  "$10"
```

then the text template becomes[2]:

[2]More accurately, we would set `waiter = person(name=>"Jeeves")` etc., and then replace variables in the text template with the value of their `name` label.

```
["Jeeves","gives","the hamburger","to","Albert"]
```

Alternatively we could use more generic names for the component's parameters:

```
waiter: "the waiter"
eater:  "the eater"
food:   "the food"
money:  "the money"
```

producing

```
["the waiter","gives","the food","to","the eater"]
```

This illustrates how generic text templates attached to generic components become automatically and appropriately instantiated in the scripts in which those components play a role.

### 3.3 Forming Variants of Detailed Scripts

Components provide the 'basic units' of assembly and disassembly of scripts. In other words, given that a script is composed of components, we now have a method by which we can *decompose* a script – namely by removing some of its components. For example, in the DCE case illustrated earlier in Figure 2, we can form alternative scripts about DCE interaction by including/excluding certain components. These script variants can be used for various tasks, such as generating natural language text or diagnosis. For example in diagnosis of computing systems, a reported timeout error would suggest including timing aspects in the script describing the observed events, whereas a reported security error would not.

### 3.4 Describing Actions from Different Viewpoints

As well as describing different phenomena, scripts can include steps describing the same phenomonon from different perspectives. For example, I might describe "putting gas in a car" as both a transfer of ownership (I become the owner of the gas, ie. the `add` list in the action includes `[Gas,owned-by,Person]`) and a movement into a container (`[Gas,in,Tank]`). Both descriptions provide different perspectives on the same event, and introduce different relations into the representation of that event. There probably needs to be some method of ensuring that we maintain particular perpsective(s) throughout the script, maybe analogous to the way compositional modelling ensures consistent modelling assumptions among fragments in a model [Falkenhainer and Forbus, 1991].

Often, multiple actions can be modelled as either a sequence of actions, or as different views of a single 'composite' action. For example, we can model `Exchange` as being a `Give` followed by a `Receive`, or alternatively as a single event involving both a `Give` and `Receive` aspect which (are modelled as) occurring simultaneously. This choice is up to the knowledge engineer, and may depend on the task at hand. The knowledge engineer may wish to construct different components (of the same realm) to represent these two alternatives, and then plug in the most appropriate component depending on the problem-solving task at hand.

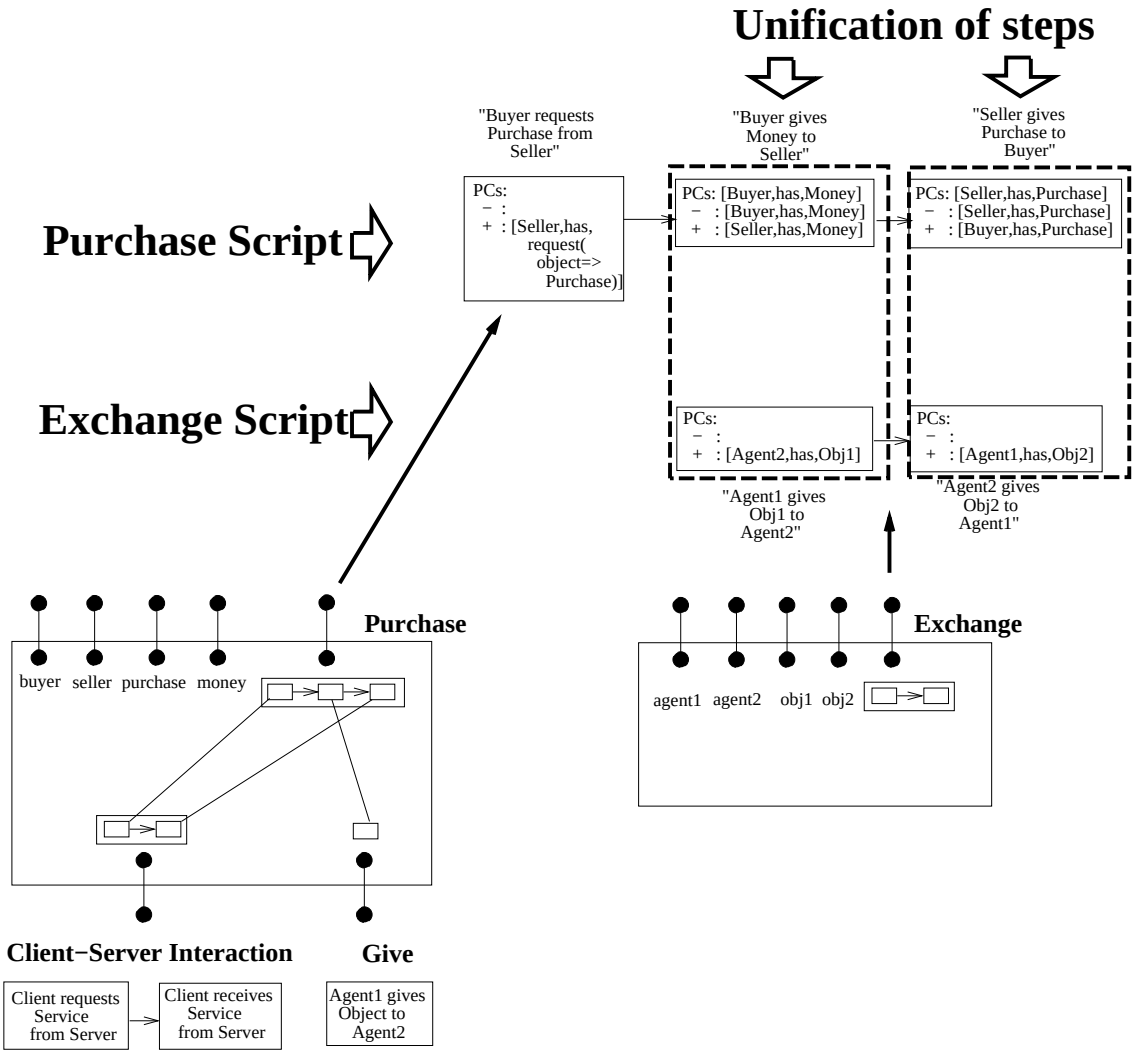



Figure 9: Identification of a new abstraction in the `Purchase` script, through unification of an `Exchange` script with it.

---

### 3.5 Identifying New Abstractions in Scripts

As well as composing a script from components, we can *inspect* that script to see if script fragments from other components will match (unify) with it. If (the script from) a component does match the larger script, then we have identified an *additional abstract pattern* in that larger script.

An example of this is shown in Figure 9. Here, we have constructed a `Purchase` script by composing components in the realms `Client-Server Interaction` and `Give`. The resulting script is the three-step sequence shown at the top of the figure. Each step is an action rule defined by preconditions, delete and add lists (these are discussed in more detail in the earlier Working Note 4), plus any additional information (eg. text skeletons) which the user might have attached to those rules.

Underneath this script in this Figure is the `Exchange` script, comprising of two steps

("Agent1 gives Obj1 to Agent2", "Agent2 gives Obj2 to Agent1"). The steps in this script, in fact, unify with steps in the `Purchase` script. By identifying this, we are able to conclude that a `Purchase` involves an `Exchange`, an abstraction not previously stated in the composition of `Purchase`. In addition, after unification, the terminals of the `Exchange` component are bound to the appropriate elements in the `Purchase` script, hence we are able to infer that *`Purchase` involves an exchange between the `buyer` and the `seller`.*

It's worth reiterating that the `Exchange` abstraction (albeit conceptually simple) was not stated in the original `Purchase` script; rather, we have been able to automatically identify that it holds through the unification process.

Note too that this unification process involves using components 'in reverse': Rather than providing the `Exchange` component's parameters and hence instantiating the `Exchange` script, we have instantiated the script first (through unification with the `Purchase` script) and as a result found out the `Exchange` parameters. This bidirectionality of components arises naturally from our use of unification as the method for structure matching. In fact, the earlier descriptions of data-flow (from a component's parameters to its sub-components, to their script fragments, and finally back to the component's script) are purely conceptual.

In this example, it so happened that the matching steps happened to be consecutive in both scripts. In general, though, a suitable matching algorithm would allow other non-matching steps to occur between steps which matched (ie. we check `script1` unifies as a subsequence of `script2`).

As a rather interesting point which the astute reader may have noticed, there is a second way in which the `Exchange` and `Purchase` script can unify, namely where the first `Exchange` step matches the *first* `Purchase` step. Thus, we can also view a `Purchase` as an `Exchange` in which the `buyer` exchanges a `request` (to buy something) for a `purchase`! This second abstraction is quite reasonable given the information provided. The knowledge engineer may later decide to modify the components to more accurately reflect his/her understanding of the concept (eg. that the exchanged objects are physical, or that the exchanged objects are of equal value, or that receipt of the objects satisfies some agent's goal etc.).

### 3.6 Describing Events from Different Viewpoints

We can use this technique of identifying new abstractions in a script to identify different *viewpoints*. This involves using the same unification method, but where the matched-in script provides a different perspective of events (eg. as reflected in it having a different text template associated with it).

An example of this is shown in Figure 10. In this example, we can match two abstractions with the script, representing client-server interaction from the client's and server's viewpoints respectively. Note that each viewpoint has different text templates attached to its actions.

Thus we could use this abstraction-identification method of Section 3.5, and/or component interchangability, to provide descriptions of the same set of events from different perspectives. For example, we could use "from the waiter's perspective" components in building a restaurant script to obtain text reflecting what the waiter would see:

```
...
I request the food from the waiter.
I receive the food from the waiter.
...
```

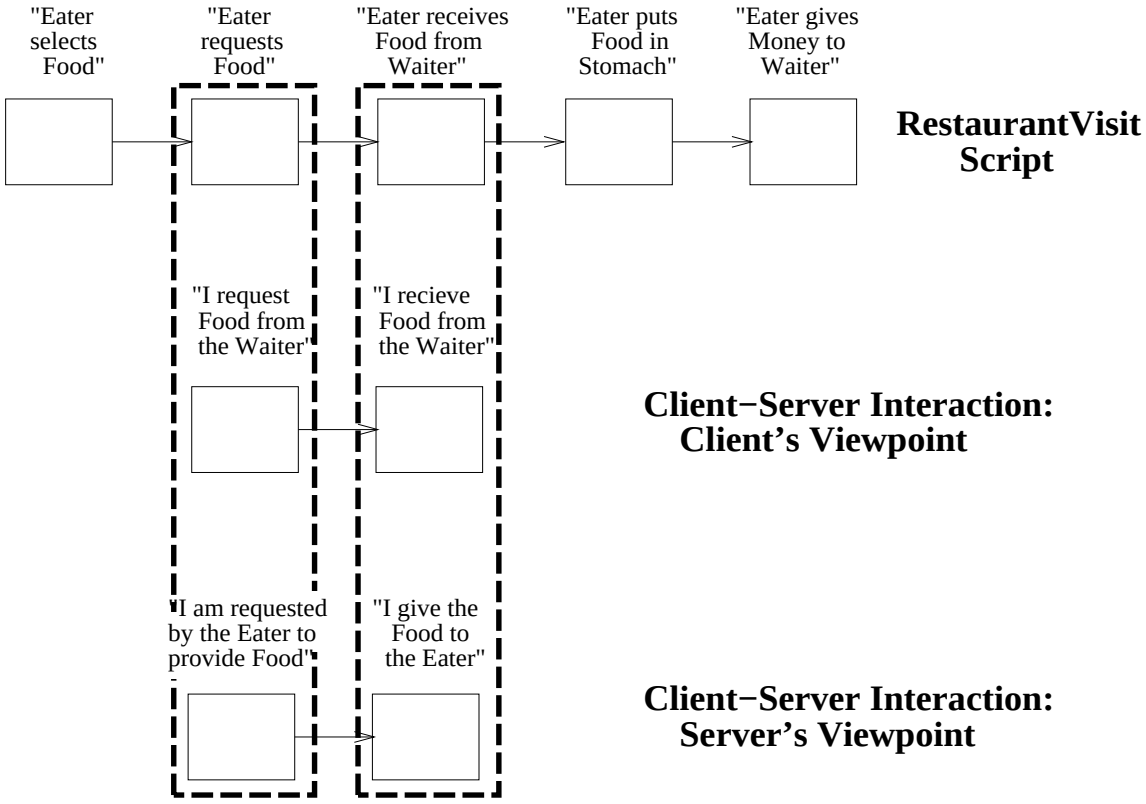



Figure 10: We can generate descriptions from different viewpoints by unifying viewpoint-specific components with the script.

Or alternatively "from the eater's perspective" components to obtain text reflecting what the eater would see:

```
...
I am requested by the eater to provide the food.
I give the food to the eater.
...
```

### 3.7 Forming Predictions/Expectations

In a similar way, we can use scripts to form *expectations* based on *partially* matching their structures with other scripts, or with a sequence of observed changes in the world (a process akin to plan recognition [Genesereth, 1982]).

Figure 11 illustrates a partial match of a `RestaurantVisit` script with observations. Two steps in the script are left unaccounted for – thus, on the assumption that this `RestaurantVisit` script is a good model of what is being observed, we can conclude that there was probably a `Eater requests Food` step earlier (which wasn't observed), and we form the expectation that the next thing we will be told is that the `Eater gives Money to the Waiter`.

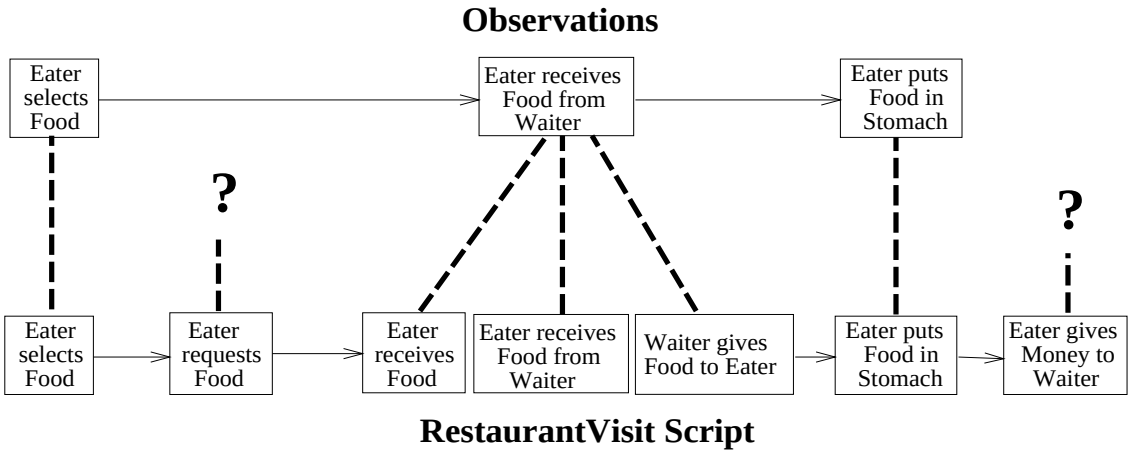



Figure 11: We can form expectations by partially matching a script with observations.

---

# 4 Comparison with GenVoca

## 4.1 Introduction

The ideas presented in this paper has been heavily influenced by work in Don Batory's Software Reuse Group at UT Austin. We have sought to apply the GenVoca model of software composition, which underlies much of that group's work, to our problem of script composition. We have found many of the GenVoca concepts can be applied to our task, but we have also deviated from certain aspects of this method. We summarize these similarities and differences here.

Good overviews of the GenVoca method are given in [Batory and O'Malley, 1992, Batory and Geraci, 1995, Batory et al., 1992]. [Batory et al., 1992, Batory et al., 1993] give examples of components and composition in the domain of data structures. P2 [Batory et al., 1993] and the object-oriented P++ [Singhal and Batory, 1993] are software generators which compose software components based on the GenVoca model. A full collection of papers from Batory's group is on the world-wide web at [Batory, 1995].

## 4.2 Similarities with GenVoca

GenVoca's premise is that by standardizing the fundamental abstractions of a domain and their programming interfaces, it is possible to build plug-compatable, inter-operable, and interchangable building blocks called *components*. Components with the same interface are said to be in the same *realm*. Components within a realm implement the same fundamental abstraction, but in different ways. Components also have sockets (referred to as "imported interfaces" or "realm parameters"), and can thus be connected. Software systems are constructed by connecting or "stacking" the appropriate components. Type equations are used to specify a particular stacking.

We have adopted all these features of GenVoca in the work presented here. In particular, the notion of a *standardized interface*, and the resulting library of *interchangable components* that this implies, has proved invaluable.

GenVoca was intended for constructing software systems, rather than the representational data structures (ie. scripts) that we are building. However, this is actually not such a big difference: a computer program can be considered a data structure (of type `program`), and conversely a script can be considered a 'program' which can be 'executed' to perform a sequence of actions (Section 3.1). The particular tasks we perform, though, on the composed programs/data-structures are typically different. In P2 and P++, composed software operations are simply executed. With scripts, though, we typically introspect on them and manipulate them in various ways for problem-solving.

In P2, a 'block' which a component exports is a piece of software implementing a particular operation. Unlike our examples which typically export a single block, P2's components export several blocks corresponding to a set of inter-related operations (eg. code implementing 'insert', 'remove', and 'get_value' operations). This allows a set of operations to be modified 'in sync', where those modifications together represent adding a single feature to the software. The component thus encapsulates code changes required to implement that feature. In the object-oriented P++, a component's parameters are classes and operations ('blocks') are grouped according to the class they belong to.

## 4.3 Deviations from the GenVoca Model

While our work is based around the GenVoca model, we have also deviated from it in a few important ways which we now outline.

### Transparancy of Blocks

In the GenVoca model, while a component can access the internal structure of its parameters, it cannot access the internals of the blocks (ie. code implementing operations) passed back up from its sub-components. The blocks are *opaque*; they are 'black boxes' which can be built on (eg. have additional code wrapped around) but not opened up.

In our work, we have instead made blocks *transparent*, allowing a component to open up, inspect, and reorganize blocks passed up from its sub-components. This is necessary if a component is to interleave block information from its sub-components, as we frequently do (eg. Figure 8).

Although P2 and P++ components never need to interleave code from their sub-components, it's possible that other software composition systems may require this ability. A simple example would occur if the language the code was written in required variable declarations to be at the start of a block, with the body of code following. In this case, if a component imported two blocks from two sub-components, it would be necessary for the component to open up those blocks, separate the variable declarations from the code bodies, and then put the variable declarations together at the start of the exported block followed by the merged code in the remainder of the exported block. This is illustrated in Figure 12.

### Structural Restrictions on Plug-compatibility

In our model of a component, each parameter has an associated *structure* (Section 2.2), corresponding to a parameter's *datatype* in GenVoca. In GenVoca, a parameter's datatype is the same for all components in a realm. In our model, however, we allow a parameter's structure/datatype to vary among components within the same realm. This allows us

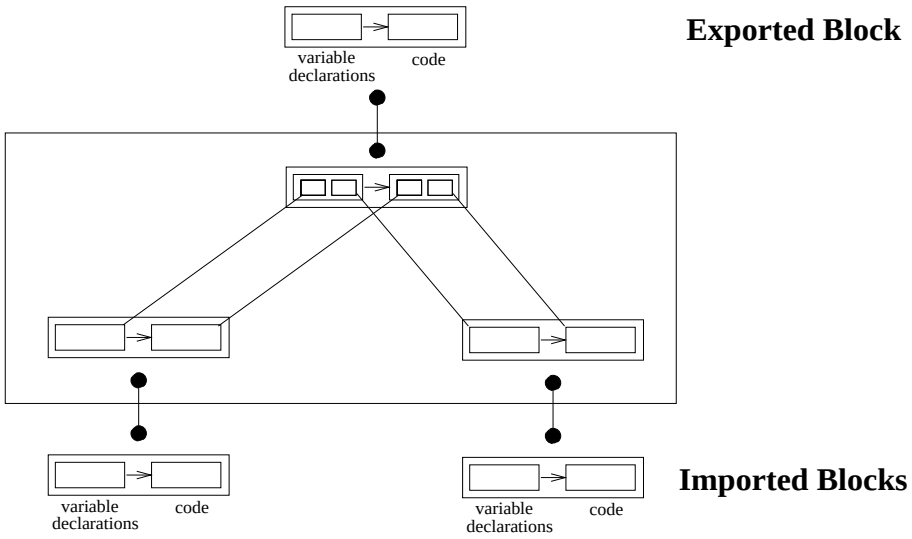



Figure 12: By making imported blocks transparent, a component can interleave information from its sub-components. Here, we ensure that variable declarations preceed the main body of code in an exported block.

---

to create special subclasses of a realm, containing components which are only applicable to specific types of data. We thus have an extended notion of plug-compatibility (Section 2.5).

It's possible this might be useful in GenVoca also: for example, in the `data-structures` realm, we might have an `Index` component which maps a nonindexed container data-structure to an indexed container, but which only operates on containers containing elements which are numbers (ie. that `Index` component exploits some particular advantages of number-containing data structures). This restriction would be reflected in our approach by restricting the `contained-element` parameter to be of type `number`. An alternative, which Batory's group has explored, might be to express such constraints in some external design-rule language, and then perform some post-hoc composition validation to ensure such constraints are not violated [Batory and Geraci, 1995].

### Input/Output Reversibility

As discussed in Section 3.5, our components have the peculiar property of being 'reversible': That is, as well as providing parameters to generate a block, we can conversely provide the block to find the parameters it used (or even provide a partial block and a few of the parameters, and find the remaining block/parameters). This property is a natural side-effect of using unification (a deterministic process) as the method of script composition. This reversibility is useful for the tasks which we wish to perform (Section 3.5), but is probably of little value for the task of software composition.

# 5 Summary

This paper has presented how a GenVoca-like approach can be used to define abstract scriptal components, and compose them together to construct more detailed scripts. In addition, a variety of other tasks can be supported through a compositional approach. We are enthusiastic about this work: First, we have finally been able to address the issue which confonted us eight months ago, namely how can we represent scripts in a compositional manner. Second, we have been able to successfully import a substantial body of ideas and concepts from software engineering into knowledge representation. Third, it seems that this approach can also be applied to composing object descriptions too (eg. `kayak`, discussed earlier in Working Note 3), where components are larger-scale abstractions such as `containership`, `floatation`, and `human-propulsion` rather than the finer-grain axioms which we considered earlier. We will explore this issue in the next working note (number 7).

# References


[Batory, 1995] Batory, D. (1995). http://www.cs.utexas.edu/users/schwartz. (collection of papers from the Predator Research Group, Dept CS, UT Austin).

[Batory and Geraci, 1995] Batory, D. and Geraci, B. J. (1995). Validating component compositions in software system generators. Tech Report 95-03, Dept CS, Univ. Texas at Austin, TX.

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

[Batory et al., 1992] Batory, D., Singhal, V., and Sirkin, M. (1992). Implementing a domain model for data structures. *Int. Journal of Software Engineering and Knowledge Engineering,* 2(3):375–402.

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

[Chapman, 1986] Chapman, D. (1986). Cognitive cliches. AI Working Paper 286, MIT, MA.

[Cullingford, 1978] Cullingford, R. (1978). Script application: Computer understanding of newspaper stories. Tech Report 116, Dept CS, Yale Univ.

[DeJong, 1979] DeJong, G. (1979). Prediction and substantiation: two processes that comprise understanding. In *IJCAI-79,* pages 217–222.

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

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

[Genesereth, 1982] Genesereth, M. R. (1982). The role of plans in intelligent teaching systems. In Sleeman and Brown, editors, *Intelligent Tutoring Systems,* pages 137–155. Academic Press, Cambridge, MA.

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

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

[Schank and Abelson, 1977] Schank, R. and Abelson, R. (1977). *Scripts, Plans, Goals and Understanding.* Erlbaum, Hillsdale, NJ.

[Singhal and Batory, 1993] Singhal, V. and Batory, D. (1993). P++: A language for software system generators. Tech Report 93-16, Dept CS, Univ. Texas at Austin, TX.