Architecture-Driven Software Component Retrieval

R. A. Riemenschneider
System Development Laboratory
SRI International
333 Ravenswood Av
Menlo Park, CA 94025
Telephone: 650-859-2507
Fax: 650-859-2844
rar@sdl.sri.com

***Abstract*— This paper argues that retrieval of complex hardware components is based on the use of architectural descriptions rather than functional descriptions, and explores the idea of basing software component retrieval on architectural descriptions. Architecture-driven retrieval is contrasted with retrieval based on keywords that characterize a few relevant properties and with deductive retrieval based on functional descriptions. A simple example, development of a component-based compiler, is used to explain how specification patterns in a architecture description language (Acme) can be used to drive component selection.**

I. Introduction

A thriving software component marketplace requires a solution to the problem of component retrieval. In a mature component marketplace, it is reasonable to expect that, while no single component will satisfy all software requirements, a properly assembled collection of specialized versions of components — perhaps supplemented by a modest amount of custom code — will do so. But, how does one find the relevant components? How does one know how to specialize them? How does one know how to assemble them? The component marketplace infrastructure must support answers to these questions.

Two basic techniques have been used for finding relevant components. The simpler technique somehow associates a set of keywords with the component. (The association can be as straightforward as choosing a descriptive component name, or maintaining a keyword-component database, but use of elementary component introspection capabilities is a popular implementation choice.) However, it seems clear that this approach doesn't scale well. Keeping the number of keywords fixed as the size of a component repository grows results in more and more components being returned in response to each query, while allowing the number of keywords to grow results in a system where components are difficult to classify and retrievals are hard to construct. In addition, unless keywords are used only within a restricted application domain, both the number of false-positives (i.e., identified components that are not actually relevant to the problem at hand) and false-negatives (i.e., relevant components that were not identified) tends to be large.

Both of the problems with the simpler technique are addressed by the more complex technique, deductive component retrieval [1], [3], [4], [8], [9], [13], [14]. In this technique, the query consists of a functional specification that the required component must satisfy. Each component has a functional specification associated with it, and a component is returned only if the required functionality is guaranteed by the component's specification. In other words, if some deduction engine can show that a component does everything it needs to do, the component is returned. Unfortunately, the deductive approach has two major shortcomings as well. First, deduction is expensive, hence relatively few components can be checked for relevance. Second, it is not necessary for a component to precisely match all the requirements to be relevant. Making the requirements relatively weak is not a solution to the latter problem, because weak requirements increases the number of false positive and, more importantly, a component can — when appropriately combined with other components — contribute to satisfaction of many requirements without fully satisfying any of them.[1]

A combination of the two techniques, using the simpler to make a rough cut and then the more complex to refine the results, can be more effective than either separately. But ultimately, as the number of components grows, the combination suffers from all the deficiencies of both techniques.

Note that neither technique makes much contribution to the problem of specializing and assembling the components returned in response to a query.

So, what reason do we have for supposing that a software component marketplace is feasible? Primarily, the fact that a healthy hardware component marketplace exists. Therefore, a

The research described in this paper was supported by SRI International Internal Research and Development funding.

[1]The later problem can be addressed in a deductive framework using Doug Smith's notion of *derived preconditions* [11], [12]. However, determining a useful derived precondition — where "useful" means, roughly, that some conjunction of functional specifications of components likely imply the precondition — is far more expensive than determining whether a component's specification implies the specification of required functionality without additional preconditions, and the latter is already too expensive for large-scale practical use.

closer look at the successful solution to the hardware component retrieval problem may prove fruitful.

II. Finding Hardware Components

For very simple hardware components, selection seems to be based on a scheme similar to the hybrid considered in the previous section. First, one makes a rough cut based based on a component category, such as *resistor* or *capacitor*, and then narrows the choices based on component functional specifications. This approach is quite effective because both the number of categories and the number of component types in each category are relatively small, and the functional specifications are quite simple.

For complex hardware components, a somewhat different selection scheme is employed. One key difference is that a complex component is characterized by the rôle it typically plays in architectures. An example should make this point clearer. To a first approximation, a resistor is characterized by the fact it is a resistor and by the amount of resistance it provides. There is no non-trivial characterization of how resistors are typically used in circuits, or how they combine with other components to provide functionality. There are no general "patterns" that provide guidance in the typical use of resistors. About all one can say is: insert a resistor when resistance is required.

Contrast this with the building of a PC. A typical sequence of design decisions might begin

1) Choose a CPU with adequate performance (say, a 700MHz AMD K7).
2) Choose an appropriate motherboard that supports adequate memory (AMD K7's use a "slot-A" type motherboard, so the choice is highly constrained.)
3) Choose a case that supplies adequate power and will house desired peripherals. ("Slot-A" type motherboards require ATX form factor cases and K7's require 300W power supplies, so again the choice is highly constrained.)

One key feature of the PC example is that the complex components are characterized, not in terms of detailed functional specifications, but in terms of the rôles they play in the PC's architecture ("CPU", "motherboard", "case", and so on), and a few key functional properties. The complete functional specification of the K7 is enormously complex, and almost entirely irrelevant; specifying that the rôle it typically plays is CPU in a PC tells us very nearly everything we need to know about it. And the selection process essentially consists of fleshing out a generic description of a PC's architecture by selecting components to play various rôles. As each component is selected, additional constraints are introduced that simplify the selection of subsequent components. The initial generic description is successively refined as each selection is made, and, eventually, a complete concrete description of the architecture to be implemented, in terms of components that can be purchased, results. This concrete description determines how every component is used, and how it contributes to the system under construction.

So, for these complex components, it is typical architectural rôles, rather than functional descriptions, that drive the selection process. These architectural rôles describe the typical use of the components. "Patterns" that provide guidance in the selection of components are easy to define. (Indeed, several are implicit in the sample sequence of design decisions.) In brief, one might say that the selection of complex hardware components is *architecture-driven*.

III. Architecture-Driven Retrieval

It is easy to see that the combination of retrieval based on keywords and deductive retrieval sketched in the introduction corresponds to the process used for selection of simple hardware components. Thus, we might expect that it is effective for libraries of simple software components, where "simple" means that the components have simple specifications, not necessarily simple implementations, and they make minimal assumptions about how they are to be used. A library of scientific subroutines is a good example. Each subroutine has a relatively short, high level mathematical specification, and the subroutines are freely combined to achieve more complex functionality. A library of standard data structures — queues, buffers, and so on — is another. The library of standard Unix utilities is yet another.

We are exploring a different approach to retrieval, inspired by the process of selecting complex hardware components outlined above. The central idea is to make retrieval architecture-driven. This means associating, with each component, not a specification of what it does on its own, but rather

- a specification of what (sub)systems built using that component, in conjunction with components that provide additional functionality, can do,
- how each of those components contributes to the overall functionality, and
- efficient procedures for finding those other components.

More specifically, one or more architecture descriptions are associated with each component. The architecture descriptions show how that component is typically used, i.e., they identify typical architecture rôles that the component plays.

As each component is selected, the associated architectural description that led us to select it is "unified" with the architectural description of the system we are building. The unification process constrains the list of candidates for filling remaining holes in the architecture. The process focuses attention on candidate components that are likely to be useful. Moreover, the selection process can be distributed in a natural way. As a result, the cost of the deduction that is required should be quite tolerable.

Revisiting the PC example should make all this clearer. (The architectural description will be somewhat simplified, since this is just an illustration.) Imagine that the architecture-driven approach to retrieval has been implemented for PC components. The architectural specification associated with a CPU might

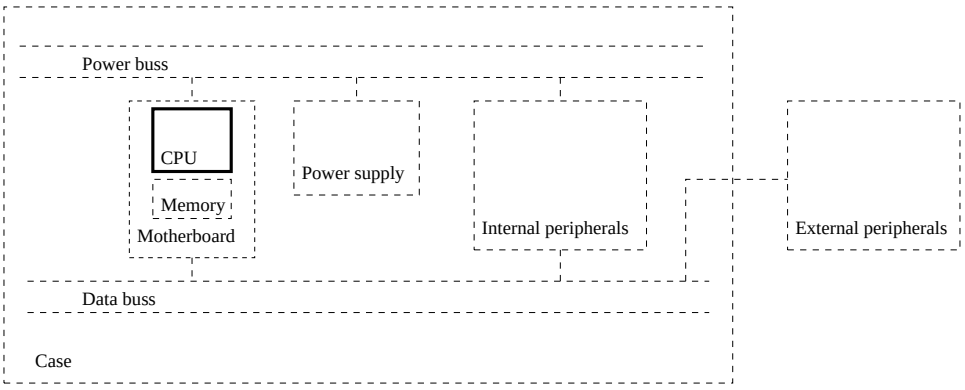

Fig. 1. CPU Architectural Specification

look something like the diagram in Fig. 1. The solid lines represent concrete components — in this case, the CPU itself is the only concrete component — and the dashed lines indicate placeholders that must be filled with concrete components to complete the system. Since the next step in building a PC after selecting a CPU is to select a motherboard, direct links from the Motherboard placeholder to candidate motherboard components are provided. Since choice of CPU directly constrains the choice of a power supply (and we assume the case includes a power supply), links to candidate case components may be supplied as well. On the other hand, choice of a CPU does not usefully constrain the choice of internal and external peripherals, links to candidate peripherals are probably not supplied.

The architectural specification of a motherboard might look something like Fig. 2. For the motherboard, we might expect links to CPUs, cases, and memory, with peripheral choices left indirectly constrained.

Once the CPU and motherboard are both selected, the architecture representation for the system under construction is shown in Fig. 3. In addition, the constraints imposed on case by the CPU (a lower bound on the power supply) and the motherboard (form factor) are conjoined to focus the next selection on a narrow range of cases: a candidate case for the system under construction must be among the CPU's case candidates and among the motherboard's case candidates.

The architectural specification of a case might look like Fig. 4. This case has two drive bays, which is reflected in the architecture by including two drives among the internal peripherals. (Of course, the case provides other constraints as well, such as choice of keyboard and mouse, but these are omitted from simplicity.) The result of unifying this case description with the partial system description in Fig. 3 is shown in Fig. 5.

Now that meaning of *architecture-driven component retrieval* is clearer, the issue of appropriate technology for achieving this objective remains. One key technology, illustrated in the example above, is provided by architecture description languages (ADLs): the ability to formally describe software architectures. In fact, our own work on transformation of architectural descriptions provides tools for dealing with descriptions that contain "variables", including match routines that can be generalized to provide unification of architectural "templates" represented in an ADL. Another key technology is provided by work on custom theorem proving systems that can be used to select appropriate components based on a description of key functional properties. Our idea is that each component will attempt to prove that it can contribute to the satisfaction of some request for functionality in the context of an architecture by finding the right partners to team with.

Basically, each component would be provided with introspective capabilities that allow it to determine, not what it can do on its won, but what it can do in conjunction with other components. In terms of the PC example, a 700 MHz AMD K7's wrapper would know it can, combined with other components, become a 700MHz PC. Thus, on receiving a request for a PC that runs at at least 633MHz, it is capable of determining that it can potentially play a role in the requested system. It will then pass on a more specific request for a PC with a 700MHZ AMD K7 to potential motherboard partners. Any motherboards that believe that are candidates will pass on an even more specific request to the cases, and so on.

The main distinctions between this approach and deductive retrieval are, first, that the deduction is more sharply focussed, and, second, that proof is considerably simplified. The focus is gained by having each component be aware of architecturally-compatible partners. Components with a fundamental architectural mismatch (e.g., motherboards that won't hold a K7, cases that won't hold a given motherboard) are never even considered. Proof simplification is achieved by reasoning about only key functional properties, such as parameters that directly affect system performance, rather than a complete functional specification. By formalizing these properties in a customized high-level modal logic of requirements, efficient resolution-based proof systems tailored to these logics can be employed. The focus on components that are likely to be relevant to the overall solution, simplified high-level component specifications, and the use of custom proof systems all contribute to making the cost of deduction more tolerable.

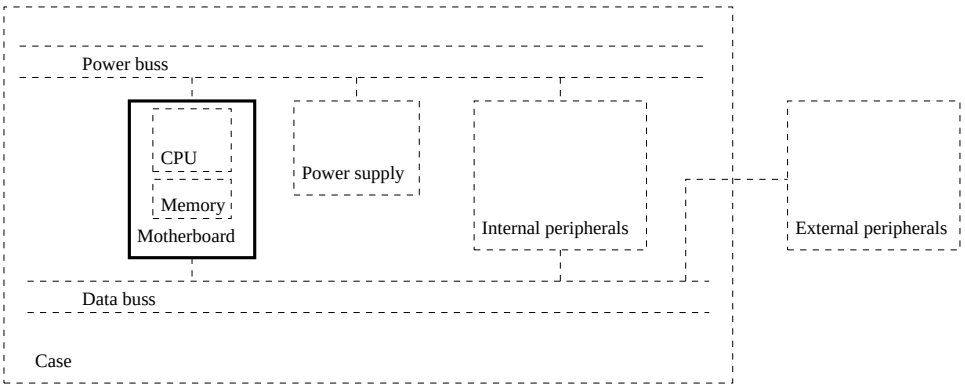

Fig. 2. Motherboard Architectural Description

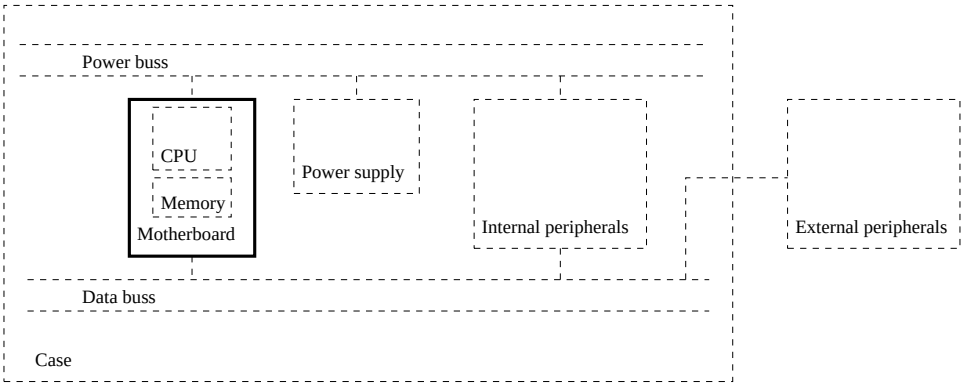

Fig. 3. CPU and Motherboard Architectural Descriptions Unified

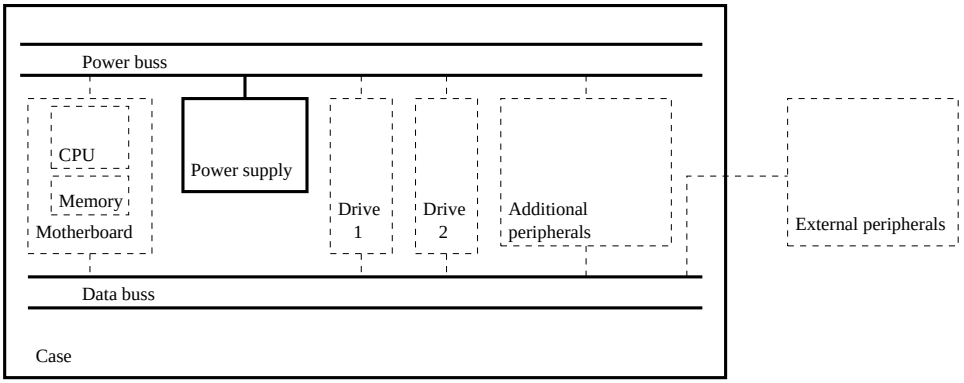

Fig. 4. PC Case Architectural Description

IV. A Software Example

Since a mature software component infrastructure does not yet exist, any example of architecture-driven component retrieval and assembly for software components is bound to be a bit contrived. However, the following somewhat artificial example will illustrate how the scheme could be made to work in a software context.

Suppose that we are assembling a compiler for some language from a stock of existing components — lexers, parsers, and so on. As a starting point, we might decide that our compiler architecture should match the pattern shown in Fig. 6, since that is what compilers built from the supposed stock of components typically look like, at a high level of abstraction. The architecture is described in a version of the Acme architecture description language [2], augmented with pattern variables that serve as place holders for as-yet undetermined elements.[2]

[2]Unfortunately, for present purposes, Acme uses 'role' to denote its solution to: X is to *connector* as *port* is to *component*. Hopefully, no confusion between roles of connectors, in this sense, and the rôles played by connectors (and components) in an architecture will result. To help distinguish between the two

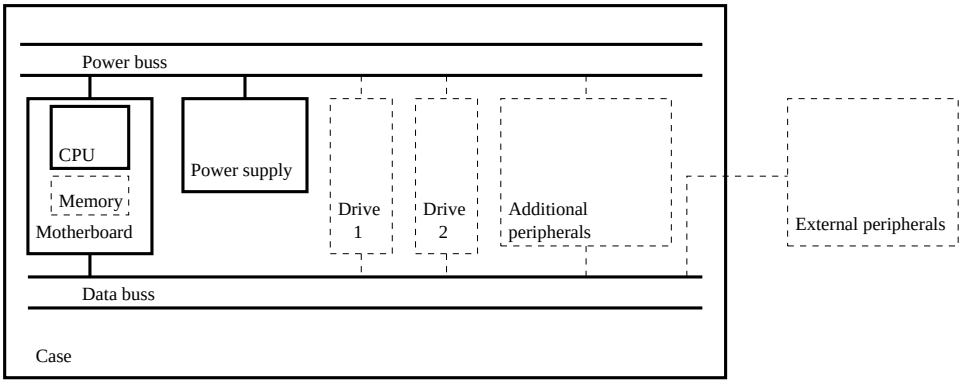

Fig. 5. Refined System Architectural Description

```
system our_compiler = {
  component @Lex = {
    ports { @LIn: character; @LOut: @Tok; @@LPorts; };
    property role_in_architecture = lexer ;
    };
  component @Par = {
    ports { @PIn: @Tok; @POut: @Parse_Tree; @@PPorts; };
    property role_in_architecture = parser ;
    };
  component @Gen = {
    ports { @GIn: @Parse_Tree; @GOut: code; @@GPorts; };
    property role_in_architecture = analyzer__optimizer__code_generator ;
    };
  connector @Lex2Par = { roles { @L2PIn, @L2POut: @Tok; @@L2PRoles; } };
  connector @Par2Gen = { ports { @P2GIn, @P2GOut: @Parse_Tree; @@P2GRoles; } };
  attachments {
    @Lex.@LOut to @Lex2Par.@L2PIn;      @Par.@PIn to @Lex2Par.@L2POut;
    @Par.@POut to @Par2Gen.@P2GIn;      @Gen.@GIn to @Par2Gen.@P2GOut;
    @@Attachments;
    };
  @@Rest
  }
```

Fig. 6. Starting Point: A Compiler Architecture Pattern

This description says that the architecture for `our_compiler` consists of (at least) three components and (at least) two connectors. One component, or combination of components, performs lexing; the pattern variable `@Lex` will be bound to this component. Note that `lexer` is simply the label of a rôle a component can play in a compiler architecture, not a shorthand for a formal definition of what lexers do. The component has two ports, whose names are undetermined, one where characters are consumed and the other where values of some undetermined type — the same undetermined type, the value of `@Tok`, that is consumed by the undetermined component performing the `parser` role in the architecture — are produced, as well as possibly other ports. (The pattern variable `@@L2Ports` will be bound to the remaining ports, if any.) The type `@Tok` is left underdetermined, because different candidates for the `lexer` and `parser` rôles in the architecture may produce and consume, respectively, different types. The pattern variables `@LIn` and `@LOut`, for "lexer in" and "lexer out", will be bound to these two ports of the lexing component. The descriptions of the parsing component, `@Par`, and the code generation component, `@Gen`, are similar, as are the descriptions of the connectors, `@Lex2Par` and `@Par2Gen`, that play the rôles of links between `lexer` and `parser` and between `parser` and `analyzer__optimizer__code_generator`. The description of the attachments of component ports to connector roles should be clear once the use of pattern variables is understood.

As a first step in constructing the compiler, suppose we select a lexer for the language of interest. The various lexer components will have been designed to play the `lexer` rôle in one or more architectures. Of course, the lexer we select must be compatible with the architectural pattern that is our starting point, in the sense that one of the architectural patterns associated with the lexer must be unifiable with the pattern for `our_compiler`. A compatible lexer pattern is shown in Fig. 7.[3] Note that use of this lexer introduces an auxiliary data structure, a symbol table, into the data structure as well. This

usages, the somewhat old-fashioned spelling of 'rôle' with a circumflex accent over the *o* will be employed only when referring to rôles in the latter sense.

[3]Definitions of the token and symbol types have been omitted, for simplicity.

```
system @Comp = {
  component lexer = {
    ports { in: character; out1: token; out2: symbol; };
    property role_in_architecture = lexer ;
    };
  component symbol_table = {
    ports { in: symbol; out: symbol; };
    property role_in_architecture = symbol_table ;
    };
  component @Par = {
    ports { @PIn1: token; @PIn2: symbol; @POut: @Parse_Tree; @@PPorts; };
    property role_in_architecture = parser ;
    };
  component @Gen = {
    ports { @GIn: @Parse_Tree; @GOut: @Code; @@GPorts; };
    property role_in_architecture = analyzer__optimizer__code_generator ;
    };
  connector l2t_pipe: pipe = { roles { in, out: symbol; } };
  connector @Lex2Par = { roles { @L2PIn, @L2POut: token; @@L2PRoles; } };
  connector @Tab2Par = { roles { @T2PIn, @T2POut: symbol; @@T2PRoles; } };
  connector @Par2Gen = { ports { @P2GIn, @P2GOut: @Parse_Tree; @@P2GRoles; } };
  attachments {
    lexer.out1 to @Lex2Par.@L2PIn;        @Par.@PIn1 to @Lex2Par.@L2POut;
    lexer.out2 to l2t_pipe.in;            symbol_table:in to l2t_pipe.out;
    symbol_table.out to @Tab2Par.@T2PIn;  @Par.@PIn2 to @Tab2Par.@T2POut;
    @Par.@POut to @Par2Gen.@P2GIn;        @Gen.@GIn to @Par2Gen.@P2GOut;
    @@Attachments;
    };
  @@Rest
  }
```

Fig. 7. One Rôle the Component `lexer` Can Play

choice of a lexer thus further constrains the choice of a parser.[4] Choosing this lexer means unifying this description with the current architecture description to obtain a new, more fully determined description of the architecture. Since the description associated with `lexer` is basically an instance of the architectural description, unification essentially replaces the initial architecture description with the description in Fig. 7, but with `@Comp` bound to `our_compiler` and `@Code` bound to `code`.

Next, suppose we select a parser. A rôle description associated with a candidate parser component is shown in Fig 8, and the result of unifying this description with the evolving description of the architecture of `our_compiler` is shown in Fig. 9. This step illustrates how our notion of architectural pattern unification involves an element of refinement: the undetermined lexer component `@Lex` in the rôle pattern for component `parser` is "bound to" a complex consisting of the connected `lexer` and `symbol_table` components of the current description of `our_compiler`, and the undetermined connector `@Lex2Par` of the `parser` description is "bound to" the pair of undetermined connectors `@Lex2Par` and `@Tab2Par` of the description of `our_compiler`.

Finally, we need to select a backend that performs analysis of the parse tree, high-level optimization, and code generation. We suppose that this is done in two stages, first selecting a component that performs that analysis and optimization, and then selecting an associated generator to obtain output of the desired type, `code`.[5] An architectural rôle description associated with the component `analyzer_optimizer` is shown in Fig. 10, the result of unifying that description with the description of `our_compiler` — note the refinement element introduced by the rôle description — in Fig. 11, and a rôle description associated with the component `code_generator` in Fig. 12. The choice of `code_generator`, and binding the remaining undetermined connectors to pipes, completes the description of `our_compiler`, as shown in Fig. 13.

Although this example has been described in terms of a series of selections by a system designer, the same series of architecture pattern unifications could result from a more agent-oriented approach. The idea is that, after selecting the initial structure for the compiler, in which the components and connections are entirely underdetermined, a bid for a compiler with this structure could be sent to various lexer agents.[6] If the lexer agent succeeds in establishing a match, it requests candidate parser

[4] Since, to keep the example as simple as possible, our Acme descriptions omit the low-level details of interfaces, this constraint simply shows up as an additional parser port. If the communication protocols were explicitly represented, the nature of the constraint — roughly, that the parser must use a certain protocol to extract certain information from the symbol table – would be clearer.

[5] We are assuming that the values of the property `role_in_architecture` are a fixed part of the component infrastructure, and that the values are ordered so as to form a meet semi-lattice. Thus, possible decompositions of underdetermined component `@Gen` in `our_compiler` is constrained by the fact that the meet of the `role_in_architecture` values of the subcomponents must be `analyzer__optimizer__code_generator`. This is one way of making decomposition inexpensive enough to consider.

[6] The exact mechanism for determining the candidates is inessential — they might register with an agent that stores the initial structural pattern, they might be obtainable by querying a central agent registry, and so on — but *some* mechanism for bounding the candidates to lexers that are likely to be able to contribute to a complete system solution is essential. Generally, asking every agent to dynamically determine whether it can possibly contribute to a solution would result in an enormous amount of wasted effort on failed unifications, even if an inexpensive check of the `role_in_architecture` values were employed.

```
system @Comp = {
  component @Lex = {
    ports { @LIn: @Char; @LOut1: token; @LOut2: symbol; @@LPorts};
    property role_in_architecture = lexer ;
    };
  component parser = {
    ports { in1: token; in2: symbol; out: parse_tree; };
    property role_in_architecture = parser ;
    };
  component @Gen = {
    ports { @GIn: parse_tree; @GOut: @Code; @@GPorts; };
    property role_in_architecture = analyzer__optimizer__code_generator ;
    };
  connector @Lex2Par = {
    roles {
      @L2PIn1, @L2POut1: token;
      @L2PIn2, @L2POut2: symbol;
      @@L2PRoles;
      }
    };
  connector @Par2Gen = { ports { @P2GIn, @P2GOut: @Parse_Tree; @@P2GRoles; } };
  attachments {
    @Lex.@LOut1 to @Lex2Par.@L2PIn1;      parser.in1 to @Lex2Par.@L2POut1;
    @Lex.@LOut2 to @Lex2Par.@L2PIn2;      parser.in2 to @Lex2Par.@L2POut2;
    @Par.@POut to @Par2Gen.@P2GIn;        @Gen.@GIn to @Par2Gen.@P2GOut;
    @@Attachments;
    };
  @@Rest
  }
```

Fig. 8. One Rôle the Component `parser` Can Play

```
system our_compiler = {
  component lexer = {
    ports { in: character; out1: token; out2: symbol; };
    property role_in_architecture = lexer ;
    };
  component symbol_table = {
    ports { in: symbol; out: symbol; };
    property role_in_architecture = symbol_table ;
    };
  component parser = {
    ports { in1: token; in2: symbol; out: parse_tree; };
    property role_in_architecture = parser ;
    };
  component @Gen = {
    ports { @GIn: parse_tree; @GOut: code; @@GPorts; };
    property role_in_architecture = analyzer__optimizer__code_generator ;
    };
  connector l2t_pipe: pipe = { roles { in, out: symbol; } };
  connector @Lex2Par = { roles { @L2PIn, @L2POut: token; @@L2PRoles; } };
  connector @Tab2Par = { roles { @T2PIn, @T2POut: symbol; @@T2PRoles; } };
  connector @Par2Gen = { ports { @P2GIn, @P2GOut: @Parse_Tree; @@P2GRoles; } };
  attachments {
    lexer.out1 to @Lex2Par.@L2PIn;        parser.in1 to @Lex2Par.@L2POut;
    lexer.out2 to l2t_pipe.in;            symbol_table:in to l2t_pipe.out;
    symbol_table.out to @Tab2Par.@T2PIn;  parser.in2 to @Tab2Par.@T2POut;
    parser.out to @Par2Gen.@P2GIn;        @Gen.@GIn to @Par2Gen.@P2GOut;
    @@Attachments;
    };
  @@Rest
  }
```

Fig. 9. Architecture of `our_compiler` After `parser` is Chosen

```
system @Comp = {
  component @Lex = {
    ports { @LIn: @Char; @LOut: @Tok; @@LPorts};
    property role_in_architecture = lexer ;
    };
  component @Par = {
    ports { @PIn: @Tok; @POut: parse_tree; @@PPorts};
    property role_in_architecture = parser ;
    };
  component analyzer_optimizer = {
    ports { in: parse_tree; out: annotated_syntax_tree; };
    property role_in_architecture = analyzer_optimizer ;
    };
  component @Gen = {
    ports { @GIn: annotated_syntax_tree; @GOut: @Code; @@GPorts; };
    property role_in_architecture = code_generator ;
    };
  connector @Lex2Par = { roles { @L2PIn, @L2POut: @Tok; @@L2PRoles; } };
  connector @Par2Ana = { ports { @P2AIn, @P2AOut: parse_tree; @@P2ARoles; } };
  connector @Ana2Gen = { ports { @A2GIn, @A2GOut: annotated_syntax_Tree; @@A2GRoles; } };
  attachments {
    @Lex.@LOut to @Lex2Par.@L2PIn;              @Par.@PIn to @Lex2Par.@L2POut;
    @Par.@POut to @Par2Ana.@P2AIn;              analyzer_optimizer.in to @Par2Ana.@P2AOut;
    analyzer_optimizer.out to @Ana2Gen.@P2AIn;  @Gen.@GIn to @Ana2Gen.@A2GOut;
    @@Attachments;
    };
  @@Rest
  }
```

Fig. 10. One Rôle the Component `analyzer_optimizer` Can Play

```
system our_compiler = {
  component lexer = {
    ports { in: character; out1: token; out2: symbol; };
    property role_in_architecture = lexer ;
    };
  component symbol_table = {
    ports { in: symbol; out: symbol; };
    property role_in_architecture = symbol_table ;
    };
  component parser = {
    ports { in1: token; in2: symbol; out: parse_tree; };
    property role_in_architecture = parser ;
    };
  component analyzer_optimizer = {
    ports { in: parse_tree; out: annotated_syntax_tree; };
    property role_in_architecture = analyzer_optimizer ;
    };
  component @Gen = {
    ports { @GIn: annotated_syntax_tree; @GOut: code; @@GPorts; };
    property role_in_architecture = code_generator ;
    };
  connector l2t_pipe: pipe = { roles { in, out: symbol; } };
  connector @Lex2Par = { roles { @L2PIn, @L2POut: token; @@L2PRoles; } };
  connector @Tab2Par = { roles { @T2PIn, @T2POut: symbol; @@T2PRoles; } };
  connector @Par2Ana = { ports { @P2AIn, @P2AOut: @Parse_Tree; @@P2ARoles; } };
  connector @Ana2Gen = { ports { @A2GIn, @A2GOut: @Parse_Tree; @@A2GRoles; } };
  attachments {
    lexer.out1 to @Lex2Par.@L2PIn;              parser.in1 to @Lex2Par.@L2POut;
    lexer.out2 to l2t_pipe.in;                  symbol_table:in to l2t_pipe.out;
    symbol_table.out to @Tab2Par.@T2PIn;        parser.in2 to @Tab2Par.@T2POut;
    parser.out to @Par2Ana.@P2AIn;              analyzer_optimizer.in to @Par2Ana.@P2AOut;
    analyzer_optimizer.out to @Ana2Gen.@A2GIn;  @Gen.@GIn to @Ana2Gen.@A2GOut;
    @@Attachments;
    };
  @@Rest
  }
```

Fig. 11. Architecture of `our_compiler` After `analyzer_optimizer` is Chosen

```
system @Comp = {
  component @Lex = {
    ports { @LIn: @Char; @LOut: @Tok; @@LPorts};
    property role_in_architecture = lexer ;
    };
  component @Par = {
    ports { @PIn: @Tok; @POut: @Parse_Tree; @@PPorts};
    property role_in_architecture = parser ;
    };
  component @Ana = {
    ports { @AIn: @Parse_Tree; out: annotated_syntax_tree; @@APorts};
    property role_in_architecture = analyzer_optimizer ;
    };
  component code_generator = {
    ports { in: annotated_syntax_tree; out: code; };
    property role_in_architecture = code_generator ;
    };
  connector @Lex2Par = { roles { @L2PIn, @L2POut: @Tok; @@L2PRoles; } };
  connector @Par2Ana = { ports { @P2AIn, @P2AOut: @Parse_Tree; @@P2ARoles; } };
  connector @Ana2Gen = { ports { @A2GIn, @A2GOut: annotated_syntax_tree; @@A2GRoles; } };
  attachments {
    @Lex.@LOut to @Lex2Par.@L2PIn;        @Par.@PIn to @Lex2Par.@L2POut;
    @Par.@POut to @Par2Ana.@P2AIn;        @Ana.@AIn to @Par2Ana.@P2AOut;
    @Ana.@AOut to @Ana2Gen.@P2AIn;        code_generator.in to @Ana2Gen.@A2GOut;
    @@Attachments;
    };
  @@Rest
  }
```

Fig. 12. One Rôle the Component `code_generator` Can Play

```
system our_compiler = {
  component lexer = {
    ports { in: character; out1: token; out2: symbol; };
    property role_in_architecture = lexer ;
    };
  component symbol_table = {
    ports { in: symbol; out: symbol; };
    property role_in_architecture = symbol_table ;
    };
  component parser = {
    ports { in1: token; in2: symbol; out: parse_tree; };
    property role_in_architecture = parser ;
    };
  component analyzer_optimizer = {
    ports { in: parse_tree; out: annotated_syntax_tree; };
    property role_in_architecture = analyzer_optimizer ;
    };
  component code_generator = {
    ports { in: annotated_syntax_tree; out: code; };
    property role_in_architecture = code_generator ;
    };
  connector l2t_pipe: pipe = { roles { in, out: symbol; } };
  connector l2p_pipe: pipe = { roles { in, out: token; } };
  connector t2p_pipe: pipe = { roles { in, out: symbol; } };
  connector p2a_pipe: pipe = { ports { in, out: parse_tree; } };
  connector a2g_pipe: pipe = { ports { in, out: annotated_syntax_tree; } };
  attachments {
    lexer.out1 to l2p_pipe.in;              parser.in1 to l2p_pipe.out;
    lexer.out2 to l2t_pipe.in;              symbol_table:in to l2t_pipe.out;
    symbol_table.out to t2p_pipe.in;        parser.in2 to t2p_pipe.out;
    parser.out to p2a_pipe.in;              analyzer_optimizer.in to p2a_pipe.out;
    analyzer_optimizer.out to a2g_pipe.in;  code_generator.in to a2g_pipe.out;
    };
  }
```

Fig. 13. Final Architecture of `our_compiler`

agents whether they can extend the match. If a parser agent succeeds, it requests candidate amalyzer/optimizer/code_generator whether they can establish a match, and so on. Whenever this process terminates in a fully determinate description of an architecture, that candidate gets returned as a solution to the original request.

V. Conclusions

While there is always ground for skepticism regarding the capabilities of untested component retrieval techniques, we have good grounds for thinking that an architecture-driven approach will prove successful. Research in agent-based systems has led to a variety of architectures, from SRI's Open Agent Architecture [5], with its centralized facilitator, to completely independent, autonomous agents. The proposed approach to retrieval can be seen as a generalization of the Open Agent Architecture that makes every agent a facilitator and that employs somewhat more sophisticated representations of the functional rôles that components play. Moreover, it is based largely on technology for architecture description and architectural pattern description that has already been implemented and proven to work effectively for both design time [6], [7] and runtime [10] transformation of architectural descriptions. The required component introspective capabilities are straightforward generalizations of capabilities in standard frameworks, such as Java Beans. However, the key question — *Just how much efficiency is gained by basing retrieval on descriptions of architectural rôles rather than functionality?* — can only be answered by experimentation.

Acknowledgements

I came up with the idea of using architecture descriptions as a basis for software component retrieval, based on the analogy with hardware, in discussions with Hassen Saïdi about how we might make use of Michael Fisher's work on custom resolution theorem provers in our component-based systems research at SRI. So, first of all, thanks to Hassen for inspiring conversation, and to Michael for crossing the pond to visit SRI and talk about his work. (Subsequent discussions with Michael made the idea of joint work on architecture-based retrieval attractive, and that fact is dimly reflected in some of the remarks about resolution theorem provers in this paper.)

Victoria Stavridou, the Director of SRI's System Development Laboratory, supported elaborating the idea, and writing this paper, with her IR&D budget, so thanks to her as well.

References

[1] B. Fischer and J. Schumann. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proceedings CADE-14 Workshop on Automated Theorem Proving in Software Engineering*, July 1997.

[2] D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language, In *Proceedings CASCON'97*, November 1997, pp. 169–183.

[3] R. J. Hall. Generalized behavior-based retrieval. In *Proceedings of the 15th International Conference on Software Engineering*, May 1993, pp. 371–380.

[4] J.-J. Jeng and B. Cheng. Using formal methods to construct a software library. In *Proceedings of the 4th European Software Engineering Conference*, LNCS vol. 717, September 1993, pp. 397–417.

[5] D. Martin, A. Cheyer, and D. B. Moran. The Open-Agent Architecture: A framework for building distributed software systems, *Applied Artificial Intelligence*, vol. 13, January–March 1999, pp. 91–128.

[6] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement, *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April 1995, pp. 356–372.

[7] M. Moriconi and R. A. Riemenschneider. *Introduction to* SADL *1.0: A language for specifying software architecture hierarchies*, Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, Menlo Park, CA, March, 1997.

[8] J. Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications*, Ph.D. Thesis, University of Cincinnati, April 1998.

[9] J. Penix and P. Alexander. Efficient specification-based component retrieval, *Automated Software Engineering*, to appear.

[10] R. A. Riemenschneider. Dependability gauges for dynamic systems. Submitted to CDSA 2001.

[11] D. R. Smith. Derived preconditions and their use in program synthesis. In *Proceedings of the 6th Conference on Automated Deduction*, LNCS vol. 138, June 1982, pp. 172–193.

[12] D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, vol. 27, no. 1, 1985, pp. 43–96.

[13] A. M. Zaremski. *Signature and Specification Matching*, Ph.D. Thesis, Carnegie Mellon University, January 1996.

[14] A. M. Zaremski and J. M. Wing. Specification matching of software components. in *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.