Relating Hierarchy Designs Back to Their Predecessors

Richard E. Courtney
Computer Science
Tulane University
New Orleans, LA 70118

David A. Gustafson
Computing and Information Sciences
Kansas State University
Manhattan, KS 66506

Abstract
In the development of medium to large software projects, the design of the system experiences several revisions. This paper presents a method to relate a design back to an earlier version by mapping the various components of the design. Each component of a hierarchy design presents a different type and amount of information that can be used in the mappings. The "effectiveness" of a component's information is reflected in a partial ordering of the sets of relationships established between the versions of a design. A relationship that contains fewer combinations of mappings is stronger than a relationship with many different mapping possibilities. The goal is to obtain a relationship that consists of only one mapping from each module in a design to its corresponding module in a previous version.

1. Introduction
This work is directed at providing a mechanism for an expert system to produce alternate hierarchy designs from an original design. The first step was to develop a representation of the holistic software design in order that transformations could be applied to alter one design to another design. This paper reports on the next step which is the examination of the changes that occur to software designs as they evolve.

Rarely is a medium to large software project designed and implemented correctly the first time. All software life-cycle models include iteration allowing a "go back and correct it" step. New, revised design documents are generally edits of the previous documents; seldom does one throw everything away and start over. The usefulness of these mappings is to determine how a design is being changed and if the pattern follows a typical software process. This is applicable in either development or maintenance.

Because a majority of the new design is identical to the old design, it is usually a relatively simple task to relate the two designs. The notion of equivalence between two designs is based on the possible implementations that could be produced from the design. Different designs can accomplish the same task. For example, one should be able to claim that the designs are externally I/O equivalent if the designs indicate the same I/O, i.e., the visible behavior of the implementations of both designs are identical.

A notion of closeness between two designs can be defined based on the similarity between the structures of the designs. For example, the control flow of two designs developed for the same Statement of Work (SOW) may not be identical, but the patterns could be very similar. Or instead of identical I/O, one design uses a menu and another uses a sequence of questions and responses.

Another use for this technique is to compare the design to the structure of the code. Although the ideal is a one to one correspondence between code and design this may not be the case for several reasons. Being able to establish a mapping between the two structures makes it easier to check if the design has been violated in "spirit" as well as textually.

We have approached the problem of being able to automate the generation of alternative designs from two viewpoints. We started by developing a set of primitive transforms. Thus, given a design, one can apply transforms to the design, to produce an alternative design. However, the primitive transforms are not particularly useful in that some operations that several CASE tools provide would take many primitive steps [3]. What is desired is a set of more complex transforms that also ensures that the design after the transform can perform the same task as the original design. Figure 1 indicates that a transform applied to a design results in a different design.

The second approach we have examined is to study the mappings between various components of a pair of designs. We recognize that a design is a collection of various elements. Modules represent the processes that
operate on the data. Data is the information communicated between the user, storage devices, operating system and the program as well as the parameters passed between modules. Subroutine calls provide the structure of the design and represent which module invokes another module. The difficult task of determining how two designs are similar is represented in Figure 2.

We have chosen to examine the mappings between components of two designs because mappings are more universal than transformations. With transformations you would have to record each operation performed on an evolving design. You would know with certainty how the revised design compares with the original, but even if a full set of transformation existed, only one aspect of the potential for comparing designs can be fulfilled. Relationships between two versions of a design would be established, but the ability to compare designs developed by different teams would not exist. By mapping components we can see how various aspects of any two designs are related. Another difference is that transforms occur to the holistic design, or entire structure, and mappings are performed on the components. Thus, even though two designs are not closely related, different mappings do exist and the designs can be compared depending upon the uniqueness of the mappings.

2. Tse's Representation

Tse developed an algebra to express software designs [5]. His endeavors revolve around the ability to manipulate designs specified with DeMarco's data flow diagrams, Yourdon and Constantine's structured design and Jackson's data directed designs. The algebra has seven sorts as shown in Table 1.

<table>
<thead>
<tr>
<th>Sort</th>
<th>Carrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>set of tasks</td>
</tr>
<tr>
<td>name</td>
<td>set of task names</td>
</tr>
<tr>
<td>struct</td>
<td>set of structures (calls from a module)</td>
</tr>
<tr>
<td>input</td>
<td>set of inputs</td>
</tr>
<tr>
<td>output</td>
<td>set of outputs</td>
</tr>
<tr>
<td>dataflow</td>
<td>set of data flows</td>
</tr>
<tr>
<td>flowname</td>
<td>set of data flow names</td>
</tr>
</tbody>
</table>

For the design in figure 3 the Tse algebra is:

```
task(A; get(); put();
  sequ(task(B; get()); put(outdata a6);
  sequ(task(H; get(source a2); put(outdata a4); elem);
  task(I; get(source a3); put(outdata a5); elem));
  iter(task(C; get(indata b2); put(outdata b3); elem));
  task(D; get(indata b4); put(sink v1); elem)))
```

Note that the structure of the system is expressed in the representation.

Some of the shortcomings of Tse's initial algebra are that a reused module has to be completely expanded, which implies that the algebra cannot express recursion. Second, it makes no mention of global data. The algebra concerned with the types of the variables.

3. Set Representation

A part of software design is decomposing a problem into manageable pieces. As we attempt to relate two design that do the same task, the comparisons made are among the pieces of the decomposition. An integral part of designs is the structure and relationships among the pieces within a design, so structure is also represented and used. To study hierarchy diagrams we have developed a mathematical definition of a software structure as sets. Our Set Representation (SR) is a seven tuple: $\Sigma = (N, D, C, M, Z, \Gamma, \iota)$.

N is the set of module names.

D is the set of data names.

C is the set of calls in the software structure, $C \subseteq \Pi \times F_i \times F_o$. A call, $c \in C$, is a three tuple, $c = (i, F_i, F_o)$. 

413
\( \Pi \) is the calling-called module pair and represents a calling arc between modules on the hierarchy diagram, \( \Pi \subseteq N \times N \).

\( P_c \subseteq D \) and is the set of parameters passed into the module. The set corresponds to the data annotation downward along the arc between modules. A call is from a parent module to a child module, therefore, the parameters indicated are the actual parameters of the parent module, not the formal parameters of the child module. The subscript \( i \) refers to the input parameters from a module; The \( o \) subscript signifies output.

\( P_o \subseteq D \) and is the set of actual parameters of the parent module returned to the parent module from the child module.

\( M \) is the set of modules in the design, \( M \subseteq N \times E_i \times E_o \times G \times F_i \times F_o \times \lambda \). A module, \( m \in M \), is an eight tuple and corresponds to a box on the hierarchy diagram.

\( n \in N \) and is the module name.

\( E_i \subseteq D \) and is the set of external inputs used in the module. \( E_i \) is the image of \( \Pi_i^E \) (\( n \)). \( E_i \) is a mapping from \( N \) to \( D^* \) and relates a name from the set of module names to an element of the powerset of data in the design. The subscript \( i \) indicates that in this case we are only interested in inputs and the superscript \( E \) says we are looking for external I/O.

\( E_o \subseteq D \) and is the set of external outputs used in the module. This is the result of the \( \Pi_o^E \) function that produces the external outputs of a module given the module name.

\( G \subseteq D \) and is the set of global data used in the module.

\( P \subseteq D \) and is the set of formal module parameters. Information about whether the parameters are passed in or passed out are contained in the call tuple and are not repeated. The use of formal parameters in the module may mean ambiguity exists between the actual parameters of the call and the names used in the module description.

\( F_i \subseteq D \) and is the set of file inputs used in the module.

\( F_o \subseteq D \) and is the set of file outputs used in the module.

\( \lambda \) is a description of the function of the module. The function is not precisely defined but "leaky" [4]. This allows the designer to change the function as he/she monitors interactions with other modules in the design. At the end of the Preliminary Design Phase, \( \lambda \) is generally not precise or formal enough for automated processing. \( \lambda \) can take several different forms.

\( Z \) is the set of data used in the design which consist of the name and the data type.

\( \Gamma \) is the set of control flows or call ordering of the software hierarchy structure. It becomes unwieldy to describe the control flow for the entire design, therefore, the control flow is described for the modules called by a particular module. \( \gamma = (n, \Omega) \) where \( n \) is the name of the module being described. As a software engineer constructs a design, he or she simulates its execution. The control flow of a design is known, but not represented in many designs. The design process continues using ill-specified control flow and module function until the designer has reduced the problem to executable pieces [1]. \( \Omega \) is the control sequencing for module \( n \) in \( \gamma \). \( \Omega \) is expressed as a context free grammar.

Figure 3 Hierarchy Diagram of Design d2b
\[ \Omega := \Psi < A > \]
\[ A := n B 1 \Omega B \]
\[ B := \alpha \]

\( \Psi \) is the control operator and is an element from the set \{sequence, selection, parallelism, iteration\}. The control operator is followed by either a list of module names or another \( \Omega \), which allows for nesting of control operations.

\( t \in N \) is the top module of the structure.

There are many ways to draw a hierarchy diagram. If two hierarchy diagrams have the same representation in this notation, they are considered to be equivalent. Any hierarchy diagram can be represented in this set notation. Order is not significant except in \( \Omega \).

4. Design Examples

In this section we present five generic designs. In the next section we will discuss observations about the various mappings.

The initial design is a basic hierarchy structure that has the top module, A, which calls three other modules, B, C, and D. Module B inputs \( a_1 \) and returns the parameter \( b_1 \). Parameter \( b_2 \) is passed down to C and \( b_3 \) is returned. C is generated. The task is called repeatedly. Module D receives parameter \( a_4 \) and \( d_4 \).

The second design of our series, \( d_2 \), also describes earlier in association with Figure 3. The letter(s) after the number indicate what modules have been expanded. In this case, module B was expanded and the design now includes modules H and I. The external input to B is removed and is somehow split between \( a_2 \) in module H and \( a_3 \) in module I. These modules both return a parameter to B whose return parameter to A has changed names to \( a_6 \).

The third design of our series, \( d_2 \), retracts the expansion of module B and expands module D. We use this design to examine how we might detect retracting effort and restarting from a previous point. The development chain in Figure 4, shows that design \( d_2 \) evolved from design \( d_1 \) and not from \( d_2b \).

The fourth design is where the afferent and transform sections of the design have been expanded. However, instead of expanding B, we added a new module \( I \) between modules A and B. Module I also calls module H. This design will be referred to as \( d_4i \).

The last design of the sequence of designs we are discussing involves the expansion of module J so that it may use module G that has already been developed.

The ordering of the different versions of our design is shown in Figure 4. The lines indicate the immediate predecessor, whereas the horizontal positioning indicates a relative time of development.

5. Mapping Components of Designs

The major concern in comparing two designs is determining how the functionality found in one design is performed in the other design. Functionality in a software
design is captured in the modules; Therefore, the relationship being established between designs is a relationship between modules. First we are going to look at the possible mappings that can occur between the different elements we have identified in a design. We will show that a partial ordering exists among the relationships we have identified using our mathematical structure by examining the information that can be abstracted from each entity and the sets of different mappings they generate.

At this time we are limiting our discussion to expansions of designs. By discussing the relationships between an expansion of a design from its parent design we are able to restrict the relationships between the two designs to mappings, or at least partial mappings, from the expanded design back to its predecessor. For example in mapping designs d2b to d1, we can assume that because a2 and a3 appear in the expansion that at least one of the inputs relates back to a1. It is not necessary that both relate back to the original because the expanded design may also reflect other changes such as error correcting or menu input in addition to command line input.

5.1. E → E Mappings

Although the concentration is with relationships between modules let us discuss the mappings between external data because of the small number of elements. The mapping of external inputs to external outputs or external outputs to external inputs uses the information of the names of the external data items. Since just the data name is available, and not the data type, there is only one element of information available to perform the mapping. Although an automated analysis of data names would be difficult, humans are good at matching what should be equivalent data elements even though the names are different. Two heuristics that assist the mapping of external data items are:

1. Map an input in the expanded design to an input in the first design if the input are the same name.
2. Map an output in the expanded design to an output in the first design if the output have the same name.

In our example designs, for design d1, Eₐ = {a1} and for design d2b or d4i, Eₐ = {a2, a3}. Mapping from the expansion back to the top level we are faced with the problem of whether one or both elements of {a2, a3} map to a1. The three possible relationships are:

(a2 → a1, a3 → a1) or (a2 → a1, a3 → e) or (a2 → e, a3 → a1).

Each relationship consists of two mappings. The first relationship says a2 maps to a1 and a3 maps to a1. The → reads maps to. The other two cases state either a2 or a3 maps to a1 and the other maps to null. The null signifies that there is no appropriate counterpart for a component in the other design.

5.2. N → N Mappings

The mapping of module names between designs uses only the names of the modules. There is little guidance to discover a "best" mapping. Heuristics of the name mappings can use guidelines from the structure of designs. These are included as the last two heuristics. The premise is that the designs follow guidelines on the number of fan-outs of a module developed by Yourdon [6] among others. The heuristics are applicable provided that the detail of the two designs are not radically different. If the later version had a module that was expanded three levels, each with an average fan-out of three, 27 modules would map to one, a violation of our heuristics. Heuristics 3 and 4 were written expecting a module to be expanded only one level at a time.

1. Map an item in the expanded design to an item in the first design if they have the same name.
2. The mapping should be an epimorphism or onto mapping. Because we are mapping from an expansion to the predecessor design everything in the predecessor should be mapped to. There should not be mappings to null. This assumes no functionality has been added.
3. If the versions do not vary greatly in the number of modules, then a module name should not map to more than 7 modules names. Fan-out should not be more than seven.
4. If the versions do not vary greatly in the number of modules, then for a module name that maps to more than one name, the average will be three modules. The average fan-out is three.

In our example d1 has modules names of: N = {A, B, C, D}. The d2b design uses the names of: N = {A, B, C, D, H, I}. The expanded design has six entries and the parent design has four module names. Since we allow a partial mapping, i.e. a module in the expansion could map to e, in this case there are 5⁶, or 15,625 possible mappings without heuristics! Below are some of the possible mappings. Those marked with an asterisk abide by the heuristics:

(A → A, B → A, C → A, D → A, H → A, I → A),
(A -> A, B -> A, C -> A, D -> A, H -> A, I -> B) ...
*(A -> A, B -> B, C -> C, D -> D, H -> B, I -> B) ...

(A -> A, B -> B, C -> C, D -> D, H -> B, I -> e),
*(A -> A, B -> B, C -> C, D -> D, H -> e, I -> B) ...
(A -> e, B -> e, C -> e, D -> e, H -> e, I -> e).

Another way of looking at this set of mappings for is the group {A, B, C, D, H, I} is mapped to the group {A, B, C, D}, symbolized as
{A, B, C, D, H, I} -> {A, B, C, D}.

The double headed arrow indicates that each element in the domain can map to any element in the range. Using the heuristics and mapping items together that have identical names and not including null, the groups become:

{A} --> {A}
{B} --> {B}
{C} --> {C}
{D} --> {D}
{H, I} --> {A, B, C, D}

This reduces the number of possible mappings to 16. Thus, the heuristics have reduced the problem of finding the best mapping by nearly three orders of magnitude!

5.3. N -> N Observations

With the exception of mapping design d2d to d2b, all the same to name mappings of a newer version to a previous version finds that the set of names for the older version is a subset of the newer version. The subset produces only one mapping: it is the size of the remaining elements that determines the number of different mappings. The number of groups created is the number of elements in the intersection of the two sets of names, which is each a name to name mapping, plus one which is all the names in the set difference of N_d2b - N_d2b mapped to N_d2b.

The mapping from d2d to d2b, which the predecessor is not a subset of the latter, requires only minor modification of the previous observations. To generalize, the number of mapping groups is the number of elements in the intersection plus 1. In this case N_d2b intersect N_d2b + 1. The intersection gives us the set of one to one groups. The remaining group is the set of new or renamed module mappings to the deleted or renamed modules of the older design.

5.4. PI -> PI Mappings

PI is the parent-child pair of the calling structure between modules in a design. This represents the arcs of a hierarchy design, a two tuple. The mappings attempt to preserve the functionality of the design. We are not concerned with being able to relate the calling arcs to each other. If we were attempting to relate PI to PI the only sensible mappings are identical PIs. PIs with no identical parent would map to e since this would most likely be an added function to the structure. PIs with the same parent in different designs would indicate either split functionality or added functionality.

There are two different methods of partitioning the modules using the information contained in the PI tuples. One we call the Node-Leaf method which divides the modules into three distinct groups. A top module which has no parents, leaf modules which have no children, and middle modules which have both parents and children. We limit the mappings to between subgroups.

Using the entire set of parent-child pairs the level of the modules is determined. This allows a partitioning of the modules into groups that are of the same level. Again we limit the mappings to between respective subgroups. We call this the Pi-Level method.

After the subgroups have been constructed, we can apply a set of heuristics to obtain a smaller set of mappings that probably still contains the best mapping.
1. Map a modules in the expanded design to a module in the earlier design if they have the same name.
2. As the mapping is being established, favor mappings from the children of module X in the expanded design to the children of module (X) in the precursor design.
3. If there is a subgroup of modules that have more than one parent, favor mappings that preserve the fan-in relation.

The set of PI is embedded in the calls. For d4i, PI = {(A, C), (A, D), (A, I), (C, F), (C, G), (D, J), (D, K), (I, B), (I, H)}. The expanded d5idj design has PI = {(A, C), (A, D), (A, I), (C, F), (C, G), (D, J), (D, K), (I, B), (I, H), (J, G), (J, L)}. An automated system can abstract information out of these sets of PI to aid in making a better mapping between the sets. The abstraction is a four tuple that consists of the module name, the number of children modules it calls, the set of parent modules that call it, and the length of the shortest path from the top module (which is the only module that does not have parents) to that module, this is the level of the module. When constructing the abstraction use the highest occurrence of a module to determine the module level.

The abstraction for the modules in d4i are: <A, 3, ø, ø>, <B, 0, [I], 2>, <C, 2, [A], 1>, <D, 2, [A], 1>, <F, 0, [C], 2>, <G, 0, [C], 2>, <H, 0, [I], 2>, <I, 2, [A], 1>, <J, 0, [D], 2>, and <K, 0, [D], 2>. The abstraction for the modules in d5idj are identical except to change or add: <G, 0, [C, J], 2>, <J, 2, [D], 2>, and <L, 0, [J], 3>. The Pi-Level method is used to form groups and some of the possible partial mapping are listed below. In this group we only use the module name, not the entire abstraction.

Zero depth: {A} --> {A}, (A --> A).
First level: {I, C, D} --> {I, C, D}, (I --> I, C --> C,
D → D) and others.


Third level: [L] → ε.

This would suggest the partial mapping to the null (L → ε). However, we could use the second heuristic to map back to the parent module (L → J) because we want to eliminate null mappings.

This set of group reduces the number of possible mappings from 285,311,670,611 in N → N without heuristics, to 7,529,536. Using the match identical name heuristic, the number of possible mappings reduces to 1, since each group except level 3 contain identical sets of module names, and saying that L → ε. However, the second heuristic indicates that module L should map to module J in the earlier design leaving us with two possible relationships, each containing 11 mappings, using this method.

The grouping obtained by the Node-Leaf method is:

No parents: [A] --> [A].

Parent(s) and children: [C, D, J, I] --> [C, D, I].


The combinations for this grouping are 1^1 * 4^4 * 7^6 or 30,118,144 possible mappings without heuristics. The Pi-Level grouping is preferred in the case of mapping an expanded design back to its predecessor. The Node-Leaf grouping would be useful in mapping between two similar but not identical designs.

5.5. Π → Π Observations

For the most part, when mapping different versions of a design, the groupings provided by partitioning modules into levels works well. A specific problem found in mapping any module name that existed in both design versions under this study, is module B which appeared in level two for designs d4i and d5idj and in level one for design d2b and d1.

A general problem of Pi-Level grouping is when one design has more levels that the other. This means at least one entire grouping maps to null, which is undesirable. The most "incorrect" relationship set obtained using Pi-Level groups in this set of versions was the mapping of d2d to d2b. Modules J and K which exist in d2d are mapped to modules H and I in d2b because they are at the same level, even though their functions are different. The rule takes precedence over the heuristics of mappings children to parents.

In the Node-Leaf groupings, a module that is expanded is changed from a leaf module to a middle module, assuming it's not the top module. If the expansion is one level then all the "new" modules appear in the leaf group. If the expansion is more than one level, "new" modules will also appear in the middle group as well as the leaf group. Thus, as in the example above, node J is in different subgroups in the different designs and cannot be mapped to itself. The exception to these observations was the addition of module I between modules A and B that was introduced in design d4i. Although it was the module B area that was expanded, module B remains a leaf node and the new module I is seen as a middle module.

5.6. C → C Mappings

The calls entities are composed of Π along with the parameters that are passed between the modules in the design. Whereas Π provided a four tuple abstraction -- <name, number-of-children, set-of-parents, level-of-module> -- calls adds four more elements, number-of-parameters-in, number-of-parameters-out, number-of-parameters-passed-down to subordinate modules and number-of-parameters-returned from subordinate modules. From this information we can use simple rules to judge a module to be afferent, efferent or transform [2]. Afferent is a module passing information into a system. Efferent is passing data out of a system. A module is transform if its task is to manipulate data. The rules to make the module type determination simply compare the counts of the parameters being exchanged in the calls.

Two elements of the set of calls for d2d are (A, B, Ø, (a61)) and (A, C, (b2), (b3)). Because C contains Π, we can use the grouping already determined by the different Π methods and further partition the modules into afferent, efferent and transform subgroups. The groups obtained in the Pi-Level method of mapping design d4i to d2d are:

{<A, 3, Ø, 0, 0, 0, 2, 2, a>} -->
{<A, 3, Ø, 0, 0, 0, 2, 2, a>}

{<1, 2, (A), 1, 0, 1, 0, 2, 1, a>} -->
{<b, 0, (a), 1, 0, 1, 0, a>}

{<d, 2, (a), 1, 1, 0, 2, 1, c>} -->
{<d, 2, (a), 1, 1, 0, 2, 1, c>}

{<c, 0, (a), 1, 1, 1, 2, 1, c>} -->
{<c, 0, (a), 1, 1, 1, 2, 1, c>}

{<b, 0, (I), 2, 0, 1, 0, 0, a>, <h, 0, (l), 2, 0, 1, 0, 0, a>} --> {ε}

{<k, 0, (D), 2, 1, 0, 0, 0, e>} -->
{<k, 0, (D), 2, 1, 0, 0, 0, e>}

{<f, 0, (C), 2, 1, 1, 0, 0, d>, <g, 0, (C), 2, 1, 1, 0, 0, d>,

{<j, 0, (D), 2, 1, 1, 0, 0, d>} -->
{<j, 0, (D), 2, 1, 1, 0, 0, d>}

The groups obtained in the Leaf-Node method of mapping d4i to d2d are:

{<A, 3, Ø, 0, 0, 0, 2, 2, a>} -->
{<A, 3, Ø, 0, 0, 0, 2, 2, a>}

{<1, 2, (A), 1, 0, 1, 0, 2, 1, a>} --> {ε}
[D, 2, (A), 1, 1, 0, 2, 1, e] -->>
[D, 2, (A), 1, 1, 0, 2, 1, e]
[C, 0, (A), 1, 1, 2, 2, e] -->> (e)
[B, 0, (I), 2, 0, 1, 0, 0, a] -->> (H, 0, (I), 2, 0, 1, 0, 0, a)
[K, 0, (D), 2, 1, 0, 0, 0, e] -->>
[K, 0, (D), 2, 1, 0, 0, 0, e]
[F, 0, (C), 2, 1, 0, 0, 0, e] -->>
[J, 0, (D), 2, 1, 1, 0, 0, e] -->>
[J, 0, (D), 2, 1, 1, 0, 0, e]

Even though there are more groups than you would have with the II mappings, the fact that there are fewer elements in each group means there are fewer possible mappings.

5.7. C → C Observations

Adding the parameter information gives us more facts to attempt a mapping between two designs. The intent is that with more information, poor mapping choices would be eliminated and there would be smaller set of relationships, which hopefully, would retain the "best" set of mappings.

The call mapping labels modules as afferent, efferent or transform further subdividing the groups created by the II mappings. A disappointing discovery is that adding parameter data results in more modules mapping to null. The null mappings are a result of the call method expecting to find afferent, efferent and transform modules on every level.

Applying the call method to the Node-Leaf groups, if the new children modules are leaf modules and are of the same afferent, efferent or transform type as their parent, they will be mapped correctly to their parent, or at least stay in the same mapping group. Otherwise, the new modules will be placed in a group that doesn't allow any mappings to the parent module. If a new module is afferent and the parent is either transform or efferent then the new module is mapped to other afferent leaf modules of the previous design, not the module it is an expansion of.

If a new module is also expanded, the module will be mapped to the same afferent, efferent or transform modules in the older design. It cannot be mapped to its parent since the parent is in the leaf group and this new module is in the middle group. The best mapping we can hope for in this case is to establish a relationship with some super-ordinate module of its parent.

5.8. M → M Mappings

The module entity is an eight tuple and can be partitioned into groups depending on whether they have any type of external I/O. Modules that use globals should be placed in a group, not only for trying to identify easier mappings, but to alert designers to where potential side effects could occur.

1. Group the modules in both designs 1 and 2 into subgroups based on whether a data set is empty or non-empty. Limit the mappings between the subgroups with identical empty, non-empty data pairings. This definition says to place into one subgroup all modules with external input, another subgroup is composed of modules with only external output, etc. Modules with both external inputs and external outputs would be in a separate group from those with only external inputs only. Recognizing the different communication methods to a module, external inputs, external outputs, globals, file inputs and file outputs, there are 2^5 or 32 different groups to place a module in. It is assumed that all modules have parameters hence that field is removed from the abstraction.

The set M for design d2b is: 
\{(A, O, O, O, O, O, O, \lambda A), (B, O, O, O, [a]O, O, \lambda B), (C, O, O, O, (b2, b3), O, O, \lambda C), (D, O, [v1], O, [b4], O, O, \lambda D), (H, \lambda B), O, O, (a4), O, O, \lambda H), (I, (a3), O, O, (a5), O, O, \lambda I), \}. The module set for design d2d, M = \{(A, O, O, O, O, O, O, \lambda A), (B, (a1), O, O, (b1), O, O, \lambda B), (C, O, O, O, (b2, b3), O, O, \lambda C), (D, O, O, O, (b4), O, O, \lambda D), (J, O, O, O, (d1, d2), O, O, \lambda J), (K, O, [v1], O, (d3), O, O, \lambda K)\}.

The abstractions from the modules do not include the \lambda entry because it is an English description of the intended function and not available for abstraction in any meaningful manner. The module abstractions obtained for design d2d and d2d and the group pairings for mapping are:

One of the possible mapping from this grouping is:

5.9. M → M Observations

With these examples the module to module mapping produced no surprises. The one item of notice is that when external I/O is added to or deleted from a module it changes the group the module belongs to. This mapping method provided the best mappings of design d2b to the
rest of the designs. Perhaps this is the best mapping heuristic for relating two designs that did not evolve from one another.

One aspect we are endeavoring to overcome is the elimination of mappings to null by establishing an order to the importance of different input and output types to a module. In the current system, if design 1 has a module with file input and an external output, and design 2 has a corresponding module with file input but not the external output, the two modules are placed in different groups and are not be paired together. We desire some relationship to be established since both modules do file input.

6. Partial Ordering of the Mappings

The previous section shows that we can map from one component of a design to corresponding components in the "original" design. However, the desire is to obtain the best reasonable set of mappings.

Mapping A is a more stringent relationship between two designs than mapping B if and only if all the mapping produced by A are also produced by B.

This definition allows us to establish a partial ordering of the information obtained from the different components of a design.

The bottom mapping method of this partial ordering is the name to name mapping without heuristics. This method produces a set of relationships where every modules name is mapped to every combinations of names in the other design.

The simple heuristic of looking at relationships where modules with identical names in the two versions are mapped to each other reduces the number of relationships significantly. Because we limited this discussion to different versions of the same design, the use of this one heuristic generally does not eliminate the relationship that contained the "best" set of mappings.

By using additional information provided by the different elements of a software design, we were able to obtain a smaller set of relationships than the N → N mappings provided. The Π → Π and M → M mappings partition the modules in different ways, therefore, one method does not provide a subset of relationships that the other generates. Even the two different methods of using the Π → Π mappings result in different mapping groups. The C → C delivers only a subset of the relationships yielded by the Π → Π. This is because Π is an element of the C tuple.

The resulting partial order of the different mapping methods is shown in Figure 5. The h subscript indicates using the identical name heuristic. That is the only heuristic shown in this diagram. A top element is missing which should be the "best" relationship between two versions and contains one mapping from each module in the newer design to a module in a predecessor design. This is because any of the methods, except N → N without heuristics could eliminate the "best" relationship.

7. Conclusions

We are beginning to develop automated tools to make comparisons between two designs. This paper discusses the relative simple task of relating different versions of a software design to each other. Side benefits of this process are insights into the development process of software design.

An automated tool has been developed to perform the mappings discussed in this paper. Its evolving design has been scrutinized by studying the mappings between the different versions. The growth of the system is similar to the generic design presented in this paper. The development of the automated system has followed a Rapid Prototype model. Therefore, new modules are continually being added to the system. These additional modules do not "correctly" map to anything in the older versions because they reflect added functionality. On the other hand, the growth of the generic design was by expansion of modules, so it was assumed the "correct" mapping of a new module would be to its parent.

Two other sets of software design versions have been studied with the system. They were developed using the Waterfall methodology. One difference noticed by examining the mappings among these sets was that only
between versions 1 and 2 were there significant changes in the number of modules. The change was an increase as expected. The majority of changes after version 2 was the addition of parameters and output messages. To accommodate this revelation, we are currently working to develop mappings between data of designs.

Other aspects that need to be resolved are effective procedures to switch a mapping of a module from null to some other module(s). Our thinking to date is to revert to the mappings established with less detailed information.

The results seen to date indicate that comparing different versions of a software design provides a method to study the software development process.

References


