Using Concept Lattices to Uncover Causal Dependencies in Software

John L. Pfaltz,

Dept. of Computer Science, Univ. of Virginia
Charlottesville, VA 22904-4740
jlp@virginia.edu

Abstract. Suppose that whenever event x occurs, a second event y must subsequently occur. We say that x "causes" y, or y is causally dependent on x. Deterministic causality abounds in software where execution of one routine can necessarily force execution of a subsequent subroutine. Discovery of such causal dependencies can be an important step to understanding the structure of undocumented, legacy code.
In this paper we describe a methodology based on formal concept analysis that uncovers possible causal dependencies in execution trace streams. We first walk through the process using a small synthetic, but easily comprehensible, example. Then we illustrate its potential using 57 threads involving 18,969 executed operations that were monitored in an open source middleware system.

1 Introduction

Since its first application as "concept analysis" [22], Galois closure [13] has proven to be a valuable tool for the analysis of various phenomena. Many examples can be found in *Formal Concept Analysis* [5] and the reader is assumed to be familiar with this fundamental work.

To our knowledge the first effort to apply closure concepts to software engineering was by Gregor Snelting who used formal concept analysis to analyze legacy code [12, 18]. Siff and Reps [17] published shortly after. Snelting's goal was to reconstruct the overall system structure by determining which variables (columns) were accessed by which modules (rows). It was hoped that the concept structure would become visually apparent as it does in all of Ganter and Wille's examples [5]. Unfortunately, the resulting concept lattice shown on page 356 of [12] is little more than a black blob. Visual interpretation of closure concepts does not seem to scale well. In [1], Ball specifically proposes using concept analysis to establish the relationship between individual test runs and procedure executions in a red-black tree system as shown in Figure 1. He, then goes on to visually identify which procedures

	Procedures					
Test	(a)dd	(1)Rotate	(r)em	(M)in	(S)ucc	(D)elFix
t1	x		x			x
t2	x	x	x			x
t3	x	x	x	x	x	
t4	x	x	x	x	x	x
t5	x	x	x	x	x	x

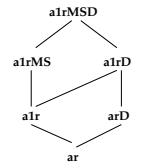

Fig. 1. Execution of sub-procedures in a red-black tree program under various test configurations.

dominate others — that is, force their execution. Given the small size of R and $\mathcal{L}$, dominance is visually derivable. But, in a larger system this might be unwieldy. Unfortunately, Ball does not seem to have done any further work on this concept based approach to dynamic software analysis. In this paper we will push this kind of analysis a bit further.

Let $a, b, c, d, e, p, q, s, t, u$ denote specific software events, and let the 8 sequences of Figure 2 depict relevant portions of trace data from 8 executions of a single software system. Our goal will be to

1	**a**	⋯	**u**	⋯	**c**	⋯	**q**	⋯	**s**										
2	**u**	⋯	**e**	⋯	**b**	⋯	**r**	⋯	**t**	⋯	**p**	⋯	**b**	⋯	**r**	⋯	**t**	⋯	**p**
3	**b**	⋯	**d**	⋯	**a**	⋯	**s**	⋯	**r**	⋯	**q**								
4	**c**	⋯	**a**	⋯	**t**	⋯	**u**	⋯	**s**	⋯	**q**	⋯	**a**	⋯	**t**	⋯	**s**		
5	**a**	⋯	**c**	⋯	**s**	⋯	**q**	⋯	**t**										
6	**d**	⋯	**e**	⋯	**c**	⋯	**t**	⋯	**s**	⋯	**u**								
7	**p**	⋯	**b**	⋯	**u**	⋯	**e**	⋯	**t**	⋯	**r**	⋯	**e**						
8	**a**	⋯	**u**	⋯	**q**	⋯	**e**	⋯	**p**										

Fig. 2. 8 event sequences extracted from simulated trace data.

analyze necessary dependencies between these events, if any. This will constitute a running example through Section 3. It provides a clearer introduction. In Section 4, we will turn to the analysis of real trace data consisting of 57 separate threads comprised of 18,969 invocations of 77 different operators.

2 Discrete Deterministic Data Mining

The first step in our analysis of software execution employs the discrete deterministic data mining (DDDM) system we have developed at the Univ. of Virginia. As described in [15, 16] this system extracts all the logical dependencies between attributes, or properties, of observed objects as recorded in a binary relation $R(O, A)$. We let $\mathcal{L}_R$ denote the concept lattice generated by $R(O, A)$. Each closed concept, together with its generator(s) determine a logical dependency. More specifically, if a subset at is a generator of the closed set $acqst$ of attributes then, as shown in [15], at logically implies $acqst$. The dependency, or implication, can be expressed in first-order notation as

$$(\forall o \in O)[(a(o) \wedge t(o)) \vee (q(o) \wedge t(o)) \;\rightarrow\; a(o) \wedge c(o) \wedge q(o) \wedge s(o) \wedge t(o)] \tag{1}$$

which by letting concatenation denote conjunction and suppressing the universal quantifier, we abbreviate as simply

$$at \vee qt \;\rightarrow\; acqst \tag{2}$$

These expressions implicitly indicate that the closed set $acqst$ has two generators. The data mining performed by the DDDM system is deterministic because these implications *must* occur. We sometimes call this closed set data mining to distinguish it from more customary *apriori*, or frequent set, data mining which yields statistical associations between the attributes, properties, or items.

The value of identifying closed sets in order to minimize redundant associations in traditional *apriori* type data mining has been rather thoroughly explored in [10, 26, 27]. To get a feel for the

power of focusing on closed sets, we observe that our DDDM system yielded 2,641 closed concepts, and thus logical implications, when applied to a relation, $R(O, A)$, consisting of 8,124 objects, or rows, and 39 attributes, or columns, that enumerated the horticultural properties of mushrooms. Admittedly, most of these 2,641 implications were either trivial or useless. Nevertheless, 2,641 closed set concepts is an order of magnitude less than the 25,210 frequent set associations generated by an open source *apriori* algorithm on the same $8,124 \times 39$ data set, using reasonable support and confidence parameters. Emphasizing Galois closed sets in data mining can have a huge performance payoff. And, a rather simple filter was able to reduce this mass of implications to only 37 relatively simple rules for determining whether a mushroom is *edible* or *poisonous*. We might consider these to be the most important attributes of any mushroom. Obtaining deterministic identification rules is very desirable here!

We say that the DDDM approach is *discrete* because the universal quantification can only be over the finite domain O comprising the relation $R(O, A)$.

The DDDM system constructs the concept lattice $\mathcal{L}_R$ incrementally in a manner that was first described by Godin and Missaoui in [7–9] and refined a bit more in [20, 21]. Incremental construction of the concept lattice facilitates incorporation of new data into an existing set of formal concepts without rereading the earlier data. The actual implementation of our system is more fully described in [16].

Given the sequence data of Figure 2 we first create the boolean relation $R(O, A)$ shown as Figure 3(a). Here (n, x) is true if event trace sequence n contains an occurrence of event x. Observe that x

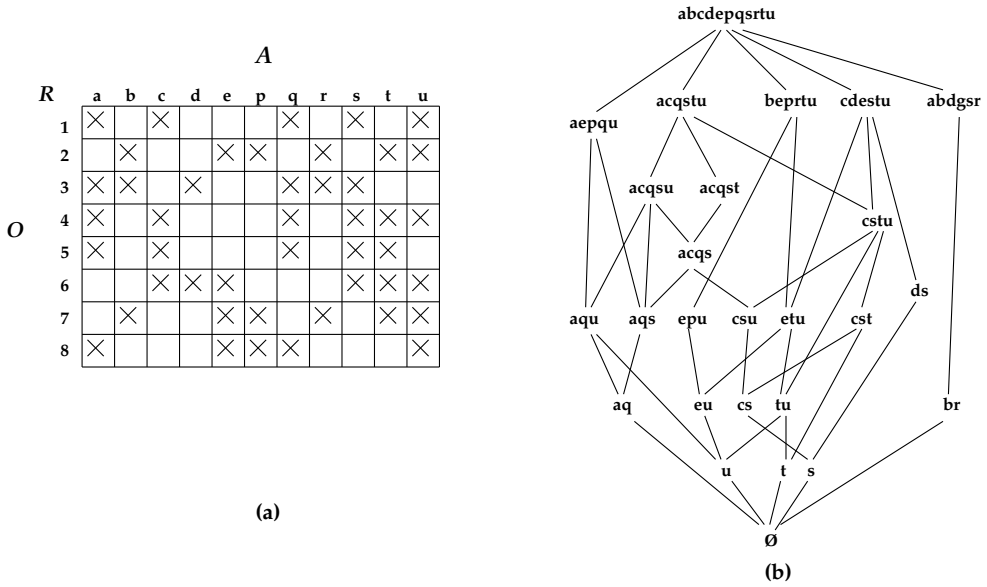

Fig. 3. The resulting concept lattice.

may, and often does, appear repeatedly in a single trace. Setting (n, x) to true only indicates that x has appeared at least once in trace n. Our DDDM system then incrementally generates the lattice of

26 concepts shown as Figure 3(b). To determine, and possibly modify, generators incrementally we use the technique proven in [11] which determines generating sets by examining covering relationships between closed concepts in $\mathcal{L}_R$. Let Z be a closed concept that covers the concepts Y_i in $\mathcal{L}_R$. If $X \subseteq Z$ is a minimal subset such that $X \cap (Z - Y_i) \neq \emptyset$ for all i, then X is a generator of Z.

Figure 4 more accurately illustrates the output generated by the DDDM process. The program does not actually draw the concept lattice $\mathcal{L}_R$; it is too hard to do well. Instead we list the attributes comprising the closed concepts, the generator(s) of these concepts, and the support of each concept (objects involved in the Galois closure), together with a concept identifier and list of covering lattice edges (which we have not shown). To conserve space, we have only listed those 20 concepts of Figure 3(b) which are supported by at least two observations. Observe that it is concept #10, *acqst* with

id#	closed concept	support	generators
1	acqsu	1,4	acu, asu, cqu, qsu
2	beprtu	2,7	bu, ru, bt, rt, be, er, bp, pr, pt
3	u	1,2,4,6,7,8	u
5	aqs	1,3,4,5	as, qs
7	br	2,3,7	b, r
9	tu	2,4,6,7	tu
10	acqst	4,5	at, qt
11	acqs	1,4,5	ac, cq
12	t	2,4,5,6,7	t
14	etu	2,6,7	et
15	ds	3,6	d
16	s	1,3,4,5,6	s
17	cstu	4,6	ctu, stu
18	csu	1,4,6	cu, su
19	cs	1,4,5,6	c
20	cst	4,5,6	ct, st
22	epu	2,7,8	p
23	eu	2,6,7,8	e
24	aq	1,3,4,5,8	a, q
25	aqu	1,4,8	au, qu

Fig. 4. Selected concepts generated by the relation $R(O, A)$ of Figure 3(a).

generators $\{at, qt\}$ that is the source of the logical implications presented earlier as (1) and (2). The reader should also verify that these logical implications are, in fact, true for the discrete domain O. For example, any object, or row, in which both a and t appear will also contain c, q and s, as indicated by concept #10.

3 Causal Dependency

Logical implication is not equivalent to causal dependency. The accepted concept of "causality" involves time, whereas logic does not. If *at* is the precedent, as in concept #10, or expression (2), then we expect that the conjunction of these two events must precede the occurrence of the events c,

q and s of the consequent if we are to say that at "causes" qst as a consequence. Causal dependence is assumed to be strictly anti-symmetric with respect to time. Since concept #10 is supported by traces 4 and 5, we examine each more closely. Event t is the last event in trace 5. In no way could it be considered to have any causal effect on the preceding events c, q or s. So similarly, the conjunction of events ct cannot possibly be a causal agent.

Now consider concept #1 which can be logically expressed as

$$acu \vee asu \vee cqu \vee qsu \;\rightarrow\; acqsu.$$

Its support is traces 1 and 4. Examination shows that, in both traces, the conjunction of events acu always precedes both q and s. So, $acu \Rightarrow qs$ is a reasonable causal hypothesis, while none of the other logical disjuncts can be. We use $\rightarrow$ to denote logical implication and $\Rightarrow$ to denote causal dependence.

We deliberately use the word "hypothesis" in the preceding sentence. We cannot establish that the conjunction of events a, c and u actually cause events q or s to occur. We can only establish that they satisfy the necessary conditions for "causality". We will discuss this further in Section 6.

Because we record the support for each closed concept along with its generators, it is not hard to re-examine the appropriate trace data sequences to verify, or exclude, specific generators as possible causal precedents. Applying this procedure to the 20 concepts of Figure 4 we get the following list of 6 possible causal dependencies shown in Figure 5.

```
#1     acu ⟹ acqsu
#7     b ⟹ r
#11    ac ⟹ qs
#15    d ⟹ s
#19    c ⟹ s
#24    a ⟹ q
```

Fig. 5. 6 possible causal dependencies.

The 6 dependencies of Figure 5 represent a rather significant reduction in the sheer number of concepts that are typically created.

It is easy to show in a first-order logic, if $a \rightarrow x$ and $b \rightarrow y$ then $ab \rightarrow xy$. Such rules of inference are common place. But, they need not be valid in causal dependence. For example, we have #19 $c \Rightarrow s$ and the trivial dependency #3 $u \Rightarrow u$. Yet, $cu \not\Rightarrow csu$ because in trace 6, $s < u$. Consequently, given the dependencies #24 $a \Rightarrow q$ and #19 $c \Rightarrow s$ we cannot logically infer that $ac \Rightarrow qs$, even though in this case #11 is, in fact, true.

Similarly, in first-order logic it is customary to declare x and y to be equivalent, $x \equiv y$ if $x \rightarrow y$ and $y \rightarrow x$. However, a concept of causal equivalence in which x causes y and y causes x does not appear to make semantic sense. Nevertheless, such apparent patterns are common in our trace data where we have repeated sections of code, or loops, as in

$$\cdots a \cdots b \cdots a \cdots b \cdots a \cdots b \cdots .$$

Such repeating patterns can be exposed by techniques developed in [24, 23], but even here, some form *a priori* knowledge of what kind of pattern is being sought must be applied.

4 A Real Example

To test these ideas, the author used trace data down loaded from *JBoss*, an open source, professional middleware company which is accessible through `www.jboss.com`. All of the method entrance events of the transaction management module in JBoss 1.4.2 were instrumented by my colleagues, Jinlin Yang and David Evans. They then ran the entire JBoss regression test suite to collect the traces [25]. A small sample of this trace data from a single thread is shown below in Figure 6

```
3    TxManager.getTransaction()Ljavax/transaction/Transaction;
2    TxManager.getThreadInfo()Lorg/jboss/tm/TxManager$ThreadInfo;
4    TxUtils.isActive(Ljavax/transaction/Transaction;)Z
1    TxManager.getStatus()I
2    TxManager.getThreadInfo()Lorg/jboss/tm/TxManager$ThreadInfo;
1    TxManager.getStatus()I
2    TxManager.getThreadInfo()Lorg/jboss/tm/TxManager$ThreadInfo;
3    TxManager.getTransaction()Ljavax/transaction/Transaction;
2    TxManager.getThreadInfo()Lorg/jboss/tm/TxManager$ThreadInfo;
4    TxUtils.isActive(Ljavax/transaction/Transaction;)Z
5    TxManager.suspend()Ljavax/transaction/Transaction;
2    TxManager.getThreadInfo()Lorg/jboss/tm/TxManager$ThreadInfo;
1    TxManager.getStatus()I
2    TxManager.getThreadInfo()Lorg/jboss/tm/TxManager$ThreadInfo;
1    TxManager.getStatus()I
2    TxManager.getThreadInfo()Lorg/jboss/tm/TxManager$ThreadInfo;
3    TxManager.getTransaction()Ljavax/transaction/Transaction;
```

Fig. 6. A representative fragment of an operator sequence

Preprocessing consisted of taking 57 such threads; scanning each operation; and, if new, assigning it an identifying integer. The integers to the left in Figure 6 are examples. This preprocessing had several benefits. First, it insures that the closed sets are extracted without using any embedded semantic information. Second, it permits us to display a set of operations as a set of integers, which we will see has definite *display* benefits. Third, because identifying integers are assigned in sequence as operators are scanned we have an interesting artifact in which related operators often appear as a number sequence. Our programs make no use of this artifact, but human inspection can reveal interesting structures that are not uncovered by the Galois closure itself.

The trace fragment of Figure 6 would be perceived by our DDDM software as

... 3 2 4 1 2 1 2 3 2 4 5 2 1 2 1 2 3 ...

From now on we consider only discrete integer representations. We observe that many operations are repeated in definite patterns, but our analysis is set based. All we can assert is that operations { 1,2,3,4,5 } all occur somewhere in this trace.

We analyzed 57 distinct traces consisting of 77 distinct operations. The shortest trace consisted of no more than 6 operations; the longest trace involved 1,393 operations.

The *set* of operations comprising each trace were input incrementally to our DDDM system. Its output is illustrated in Figures 7 and 8.

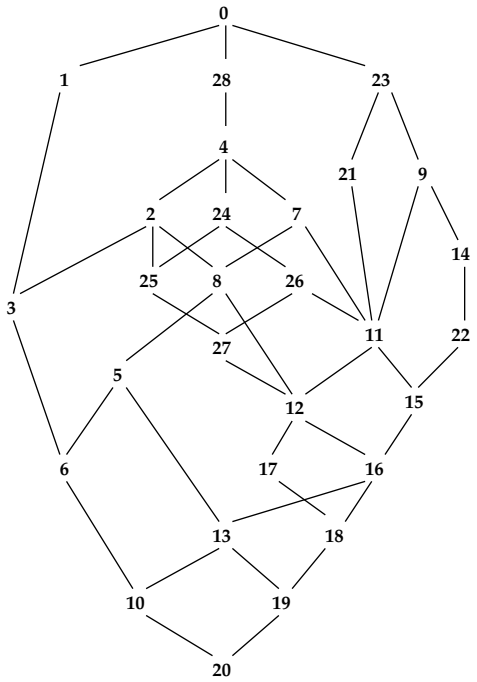

Fig. 7. Lattice, $\mathcal{L}_{ops}$, of closed operator sets.

We were more than a little surprised. Only twenty seven non-trivial closed sets of operations emerged.

As we indicated in Section 2, the lattice shown in Figure 7 is hand drawn. For each concept in the lattice our software really outputs a sequence of concepts consisting of the items in the closed set of the concept, the set of generating sets, the set of supporting rows and the set of concepts covered by the concept. The latter facilitates drawing the lattice and maneuvering through the lattice. This table of closed concepts, in Figure 8, and their generators requires a bit of interpretation; particularly since we use the hyphen (-) in two different ways.

The first column denotes the concept number. They correspond to the concept numbers in Figure 7. The reader can verify that the closed sets of concepts in the lattice below any specific concept, say concept #2, are contained in the closed set of #2. The closed set of concept #2 consists of operations 1 through 5 and operations 12 through 64. Here, rather than enumerating every operation id, separated by commas, we use the hyphen as an "extended and".

Concept #2 has many generators. Operations 33 and 34 are each singleton generators. If either operation occurs in a trace then every operation of the closed set must also occur. The combination { 4, 12-32 } is also listed as a generator. This means that { 4, 12 }, { 4, 13 } ... { 4, 32 } are each generating sets. Here we are using the hyphen as an "extended or", that is, operation 4 in combination with operation 12 or operation 13 or ... is a generator.

Concepts #17 and #18 are of interest because all of their generators are singleton; but there are many of them. For example, from concept #17, if any of the operations 46 through 49 appear in the trace then all of the operations in the closed set will occur in the trace.

Concept number	Closed set	Generators	Size of support
1	{1-11}	{6-11}	1
2	{1-5, 12-64}	{33-34},{4,12-32},{4,35-64},{1,25-32},{1,45}	6
3	{1-5}	{4}	7
4	{1-5,12-65}	{4,65},{33-34,65},{1,25-32,65},{1,35-38,65}, {1,45,65}	4
5	{1-3,5,13-14}	{1,13-14}	27
6	{1-3,5}	{1}	28
7	{1-3,5,12-24,39-44,46-65}	{1,65}	10
8	{1-3,5,12-24,39-44,46-64}	{1,12},{1,15-24},{1,39-44},{1,45-64}	12
9	{2-3,5,12-24,39-44,46-72}	{46-49,66-72},{51-52,66-72},{60,66-72}, {62,66-72}	7
10	{2-3,5}	{5}	56
11	{2-3,5,12-24,39-44,46-65}	{46-49,65},{51-52,65},{60,65},{62,65}	23
12	{2-3,5,12-24,39-44,46-64}	{5,46-49},{12,46-49},{5,51-52},{12,51-52}, {5,60},{12,60},{5,62},{12,62}	25
13	{2-3,5,13-14}	{5,13-14}	55
14	{2-3,5,12-24,39-44,50, 53-59,61,63-72}	{66}	10
15	{2-3,5,12-24,39-44,50, 53-59,61,63-65}	{65}	27
16	{2-3,5,12-24,39-44,50, 53-59,61,63-64}	{12},{5,16-24},{5,39-44},{5,50},{5,53-59} {5,61},{5,63-64}	29
17	{2-3,13-24,39-44,46-64}	{46-49},{51-52},{60},{62}	26
18	{2-3,13-24,39-44,50, 53-59,61,63-64}	{15-24},{39-44},{50},{53-59},{61},{63-64}	30
19	{2-3,13-14}	{13-14}	56
20	{2-3}	{}	57
21	{2-3,5,12-24,39-44,46-65, 73-75}	{73-75}	2
22	{2-3,5,12-24,39-44,46-72}	{67-72}	11
23	{2-3,5,12-24,39-44,46-75}	{66-72,73-75}	1
24	{2-3,5,12-32,35-65}	{25-32,65},{35-38,65}	5
25	{2-3,5,12-32,35-64}	{25-32},{35-38}	7
26	{2-3,5,12-32,39-65}	{45,65}	6
27	{2-3,5,12-32,39-64}	{45}	8
28	{1-5,12-65,76-77}	{76-77}	1

Fig. 8. Output from DDDM system.

The final column displays the number of traces in which this concept can be found, that is its size of "support". We felt that enumerating this support, as in Figure 4 would be overkill.

It is interesting to note that the infimum set { 3, 2 } of operations (concept #20) is generated by the empty set. These two operations occur in every trace.

5 Establishing Dominance

Execution of a procedure, or operator, often depends on a conjunction of conditions. Figure 8 lists many such conjunctive generators. However, analysis of such generators is beyond the capabilities of our current software. Consequently, we will restrict ourselves to analyzing only those concepts with singleton generators. Our DDDM software makes identification of these concepts quite easy.

If a singleton set, such as { 1 }, generates the Galois closed set { 1, 2, 3, 5 } as in concept #6 of Figure 8 then we can logically assert that

$$1 \rightarrow 1235 \tag{3}$$

or equivalently, "if 1 appears in a trace then 2, 3, and 5 must also appear". But, this does not necessarily imply that the execution of operation 1 "causes" the execution of the other operators. There need be no causal dependency. Such logical implication is only one necessary condition

A second necessary condition for causal dependency is that the generator must precede the consequent(s) in *all* traces. In the 28 traces supporting concept #6 this is not always true, so (3) cannot be rewritten as a causal dependency.

For each singleton generator, $\{gen\_op\}$, of a concept, we re-examine each of the trace sequences supporting the concept. If an operator *op* in the closed set of the concept precedes the first occurrence of *gen_op* in *any* trace, then $gen\_op \not\Rightarrow op$. If *op* always follows at least one occurrence of *gen_op* then we say that *gen_op* dominates *op* and $gen\_op \Rightarrow op$ becomes plausible.

This reasoning is particularly applicable when there are several singleton generators. They cannot all be causally equivalent. In the following analysis procedure we first resolve domination among multiple singleton generators, if any, and then analyze domination among the other operators of the closed concept set.

```
operator_domination (LATTICE L, TABLE Dominates)
 // Analyze the concepts in 'L' to create the
 // table of operator domination
 {
 ELEMENT      dg, fg,      // generators -- dominating, first
              o;
 SET          SG, OP;
 CONCEPT      c;
 OP_SEQUENCE seq;

 for_all c in L.concepts do
      {
      SG <- singletons (c.generators);
      if empty(SG)
           continue;
                          // concept 'c' has singleton generators
      dg <- null;         // as yet no dominating generator
      for_all seq in c.support do
           {     // examine every sequence supporting this concept
                 // get the first singleton generator in this sequence
           fg <- first_element (SG, seq);
```

```
            if dg = null
                dg <- fg;      // now, check that 'fg' is always first
            else
                dg <- null;
                break;
            }
        if dg = null            // no generator in SG is dominating
            continue;
        else
            add [dg -> SG] to 'Dominates';
                 // now, check if 'dg' dominates other operators in 'c'
        OP <- c.closed_set not c.SG;
        for_all o in OP do
            {     // verify that this 'o' follows 'dg' in all
                  // supporting sequences
            for_all seq in c.support do
                {
                if o precedes dg in seq
                    {
                    OP <- OP not {o};
                    break;
                    }
                }
        if not empty(OP)
            add [dg -> OP] to 'Dominates';
        }
    }
```

Application of this `operator_domination` procedure to the output of our DDDM software, as illustrated in Figure 8, is shown in Figure 9. We observe that concepts #3, #6, #10, #27 make no contribution to this list of possible causal dependencies, even though all have singleton generators. They fail the operator dominance test.

There are numerous instances of multiple generator sequences such as

$$46 \Rightarrow 47 \Rightarrow 48 \Rightarrow 49$$

in concept #17. We surmise that these traces come from software employing a stack architecture and that these represent iterated invocations down through the stack. Concept #18 gives rise to many apparent dependencies. Readily we could simplify the enumeration considerably. Applying causal transitivity we could simply write, for instance,

$$16 \Rightarrow 17 \Rightarrow \{24, 56\}$$
$$18 \Rightarrow 19 \Rightarrow \{24, 39\}$$

Events 24, 39 and 56 in turn generate more of the closed set comprising concept #18. Such condensation makes certain differences more evident, but our preliminary, proof of concept software is not, as yet, capable of doing it.

```
#1     6 ⟹ 7 ⟹ 8 ⟹ 9 ⟹ 10 ⟹ 11
#2     33 ⟹ 34 ⟹ { 4, 35 ... 38 }
#14    66 ⟹ { 24, 50, 53 ... 59, 61, 63, 64, 67 ... 72 }
#15    65 ⟹ { 24, 50, 53 ... 59, 61, 63, 64 }
#16    12 ⟹ { 13 ... 15 }
#17    46 ⟹ 47 ⟹48 ⟹49 ⟹ { 51, 52, 64 }
#18    16 ⟹ 17 ⟹ { 24, 50, 53 ... 59, 61, 63, 64 }
       18 ⟹ ,.. ⟹ 23 ⟹ { 24, 39 ... 44, 50, 53 ... 59, 61, 63, 64 }
       24 ⟹ { 50, 53 ... 55, 58, 59, 61, 63, 64 }
       39 ⟹ ,.. ⟹ 44 ⟹ { 50, 53 ... 59, 61, 63, 64 }
       50 ⟹ {53 ... 55, 58, 59, 61, 63, 64 }
       53 ⟹ 54 ⟹ 55 ⟹ {61, 63, 64 }
       56 ⟹ 57 ⟹ {61, 63, 64 }
       61 ⟹ { 63, 64 }
#19    13 ⟹ 14
#21    73 ⟹ 74 ⟹ 75 ⟹ { 24, 46 ... 73 }
#25    25 ⟹ ,.. ⟹ 32 ⟹ { 35 ... 64 }
#28    76 ⟹ 77
```

Fig. 9. Possible causal dependencies.

6 Considerations

The technique described in the preceding sections has four distinct steps. It: (1) identifies software events of interest; (2) extracts them from trace data to form a relation $R(T, E)$[1]; (3) creates a concept lattice $\mathcal{L}_R$ embodying a number of logical implications of the form $<$ *generator* $>\rightarrow<$ *closed concept* $>$; and (4) retains only those implications for which the $<$ *generator* $>$ precedes the remainder of the consequent $<$ *concept* $>$ in all supporting trace sequences in T. This approach works. But, there are still a number of issues to be considered.

First, the prior identification of software events of interest can be awkward. If the events denote entrance, and exit, from modules, procedures or other bodies of code as in Ball [1] and this paper, then this step is fairly straight forward. But, there are other kinds of "events" that are of interest in software analysis. Prime examples are "conditions" such as "$x + y > 100 * z$". Typically such conditions form the basis of triggers, or guards. Uncovering the various relationships between conditions and the events they may trigger is a key to finding the "likely invariants" that describe a body of software [1, 3, 14].

Michael Ernst, in particular, has been a leader in identifying likely invariants from dynamic trace data [3, 14]. Causal dependencies are a form of software invariant. So, this paper can be considered to be an extension of his work. But, neither Ernst nor we know how to discover what conditional relationships might participate in a likely software invariant without first identifying them *a priori*. It is a significant outstanding problem that we are currently investigating.

Second, given a set of causal dependencies such as Figure 5 we would like to be able to reason about them. Some rules, such as the transitive law, $if\ x \Rightarrow y\ and\ y \Rightarrow z\ then\ x \Rightarrow z$ remain true in a causal logic. But, as we saw in Section 3, others do not.

[1] It seems appropriate in this application to relabel the relation $R(O, A)$ as $R(T, E)$, where T denotes the set of traces and E denotes the set of events.

There is a considerable body of literature concerning "temporal logic" which has been studied since the early 70's as an analytic tool associated with finite state controllers, reactive devices and parallel systems [2, 4, 6, 19]. Most varieties of linear time logic (LTL) introduce 4 additional temporal operators, X, U, F, G where, given boolean expressions α, β, we have

$X\alpha$ denotes "next α"
$F\alpha$ denotes "eventually α"
$G\alpha$ denotes "generally (or always) α".
$\alpha U \beta$ denotes "α until β"

Causal dependencies can be expressed in terms of the X and F operators. Unfortunately valid derivations within a temporal logic are rare. Temporal logics encounter the same kinds of issues we have illustrated with our causal dependencies.

The third issue we must consider is the last step, `operator domination`, in which we winnow out those logical implications which cannot represent causal dependencies. As was pointed out in Section 3, we can use the support for each concept to limit the number of trace sequence that must be examined to verify the temporal precedence properties. But, this would seem to negate much of the advantage obtained by incrementally creating the concept lattice in the manner of Godin and Missaoui. We will have to keep the entire set of trace data on hand and possibly re-examine hundreds of trace sequences as each new concept is entered into the lattice.

Fortunately, this is only an apparent problem caused by our rough and ready, proof of concept software. One can incrementally create a "precedes" relation, $<$, as shown in Figure 10. Here, $x < y$ if

<	a	b	c	d	e	p	q	r	s	t	u
a			×		×	×	×		×	×	×
b	×			×	×	×	×	×	×	×	×
c	×						×		×	×	×
d	×		×		×		×	×	×	×	×
e		×	×			×		×	×	×	×
p		×			×			×		×	×
q					×	×		×		×	
r						×	×			×	
s							×	×			
t						×	×	×	×		×
u	×	×		×	×	×	×	×	×	×	

Fig. 10. The precedes relation from Figure 2.

x precedes y in any trace t. Readily, $<$ is only a pre-order, since it is transitive but not antisymmetric. If $x < y$ then y cannot causally determine x. By creating a precedence relation in parallel with the concept lattice, one can incrementally uncover likely causal dependencies on the fly with no need to re-examine earlier trace data, or even to retain it. Of course, if we do not keep the original trace data we will then lose the opportunity to look more carefully at any particular trace to see why this is, or is not, a case of causal dependency.

But possibly a precedence relation such as Figure 10 can do more. Why not use this relation to directly indicate causal dependencies? For example, we see in Figure 10 that $b < a$ while $a \not< b$. Thus we know that b precedes a in at least one trace, and that a never precedes b. This strict anti-symmetry is one property that we have postulated for causal dependence. It is a necessary condition. But, it is not sufficient. Our understanding of causal dependence $b \Rightarrow a$ is that whenever b occurs then a must also always follow. This is not true in the sequences 2 and 7 of Figure 2. This second

necessary condition whose logical expression is

$$(\forall t \in T)[b(t) \rightarrow [b(t) < a(t)]] \tag{4}$$

seems to be a fundamental property of causal dependence that cannot be derived from simple precedence relations.

This author believes that the key to discovering causal dependencies from observed software behavior must involve the use of Galois closure, which is the basis of formal concept analysis. Only by adopting a formal concept methodology can we derive an expression such as (4). It seems to have been a key piece that has been missing in the search for "likely software invariants".

Finally, we observe that our procedure still only reveals "likely" causal dependencies. Because we find that $a \Rightarrow q$ in the set T of trace data, we cannot literally say that the event a "causes" the event q as a consequence. One can only base such a claim on examination of the code itself. But, without having likely dependencies to specifically look for, such examination is extremely difficult; and in the case of legacy systems without source code it is essentially impossible.

The principles of formal concept analysis have an important application in software analysis and software engineering.

References

1. Thomas Ball. The Concept of Dynamic Analysis. *Proc. Seventh European Software Engineering Conf.*, pages 216–234, Sept. 1999.
2. E. Allen Emerson. Temporal and Modal Logic . In *Handbood of Theoretical Computer Science*, pages 997–1071. Elsevier Science, 1990.
3. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Software Eng.*, 27(2):1–25, Feb. 2001.
4. Michael Fisher. A Model Checker for Linear Time Temporal Logic. *Formal Aspects of Computing*, 4(3):299–319, 1992.
5. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis - Mathematical Foundations*. Springer Verlag, Heidelberg, 1999.
6. Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly Automatic Verification of Linear Temporal Logic. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proc. of 15th IFIP Workshop*, pages 3–18, Warsaw, Juue 1996.
7. R. Godin and Hafedh Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. In *ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, pages 394–410, Washington, DC, 1993.
8. Robert Godin and Rokia Missaoui. An Incremental Concept Formation Approach for Learning from Databases. In *Theoretical Comp. Sci.*, volume 133, pages 387–419, 1994.
9. Robert Godin, Rokia Missaoui, and Hassan Alaoui. Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices. *Computational Intelligence*, 11(2):246–267, 1995.
10. Karam Gouda and Mohammed J. Zaki. Efficiently Mining Maximal Frequent Item Sets. In *1st IEEE Intern'l Conf. on Data Mining*, San Jose, CA, Nov. 2001.
11. Robert E. Jamison and John L. Pfaltz. Closure Spaces that are not Uniquely Generated. *Discrete Appl Math.*, 147:69–79, Feb. 2005. also in Ordinal and Symbolic Data Analysis, OSDA 2000, Brussels, Belgium July 2000.
12. Christian Lindig and Gregor Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *Proc of 1997 International Conf. on Software Engineering*, pages 349–359, Boston, MA, May 1997.

13. Oystein Ore. Galois Connexions. *Trans. of AMS*, 55:493–513, 1944.
14. Jeff H. Perkins and Michael D. Ernst. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. *Proc. SIGSOFT'04/FSE-2*, pages 23–32, Nov. 2004.
15. John L. Pfaltz and Christopher M. Taylor. Closed Set Mining of Biological Data. In *BIOKDD 2002, 2nd Workshop on Data Mining in Bioinformatics*, pages 43–48, Edmonton, Alberta, July 2002.
16. John L. Pfaltz and Christopher M. Taylor. Concept Lattices as a Scientific Knowledge Discovery Technique. In *Workshop on Discrete Mathematics and Data Mining, 2nd SIAM International Conference on Data Mining*, pages 65–74, Arlington, VA, Apr. 2002.
17. Michael Siff and Thomas Reps. Identifying Modules via Concept Analysis. In *Intn'l Conf. on Software Maintenance*, pages 170–179, Bari, Italy, Oct. 1997.
18. Gregor Snelting and Frank Tip. Reengineering Class Hierarchies Using Concept Analysis. In *Proc. ACM SIGSOFT 6th International Symposium on Foundations of Software Engineering, FSE-6*, pages 99–110, Lake Buena Vista, FL, 1998.
19. Paulo Tabuada and George J. Pappas. Linear Time Logic Control of Discrete-time Linear Systems. *IEEE Trans. on Automatic Control*, page (in revision), 2005.
20. Petko Valtchev, Rokia Missaoui, and Robert Godin. A Framework for Incremental Generation of Frequent Closed Itemsets. In Peter Hammer, editor, *Workshop on Discrete Mathematics & Data Mining, 2nd SIAM Conf. on Data Mining*, pages 75–86, Arlington, VA, April 2002.
21. Petko Valtchev, Rokia Missaoui, Rouane Hacene, and Robert Godin. Incremental Maintenance of Association Rule Bases. In *Proc. Workshop on Discrete Mathematic and Data Mining*, San Francisco, CA, 2003.
22. Rudolf Wille. Restructuring Lattice Theory: An approach based on hierarchies of concepts. In Ivan Rival, editor, *Ordered Sets*, pages 445–470. Reidel, 1982.
23. Jinlin Yang and David Evans. Automatically Inferring Temporal Properties for Program Evolution. In *15th IEEE Symposium on Software Reliability Engineering (ISSRE 2004)*, Saint-Malo, France, Nov. 2004.
24. Jinlin Yang and David Evans. Dynamically Inferring Temporal Properties. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2004*, Washington, DC, June 2004.
25. Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Terracotta: Mining Temporal API Rules from Imperfect Traces. In *28th Internl. Conf. on Software Engineering (ICSE 2006)*, page (submitted), Shanghai, China, May 2006.
26. Mohammed J. Zaki. Generating Non-Redundant Association Rules. In *6th ACM SIGKDD Intern'l Conf. on Knowledge Discovery and Data Mining*, pages 34–43, Boston, MA, Aug. 2000.
27. Mohammed J. Zaki and Ching-Jui Hsiao. CHARM: An Efficient Algorithm for Closed Association Rule Mining. In Robert Grossman, editor, *2nd SIAM International Conf. on Data Mining*, pages 457–473, Arlington, VA, April 2002.