Logical Implication and Causal Dependency

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 first uncovers necessary logical implications between software events, and then extracts possible causal dependencies from execution trace streams. We provide an example of its potential by applying our method to 1,227 threads involving 498,489 executed events that were monitored in a well-known open source middleware system.

1 Introduction

Since its first application as "concept analysis" [18], Galois closure [9] has proven to be a valuable tool for the investigation of various phenomena. Many examples can be found in *Formal Concept Analysis* [3] and the reader is assumed to be familiar with this fundamental work. Our past interest in Galois closure and concept analysis has been its ability to extract logical implications in observed data. But, until now we had only been concerned with biological applications [12, 13]. Our goal in this research has been to use concept analysis to discover likely causal dependencies in software from execution traces without requiring any *a priori* semantic information.

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 [8, 15]. Siff and Reps [14] 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 [3]. Unfortunately, the resulting concept lattice shown on page 356 of [8] 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 retrieval system. However, Ball does not seem to have done any further work on this concept based approach to dynamic software analysis. The conclusion we will draw in this paper is that concept based software analysis has considerable promise, much of which is as yet unexplored.

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 [12, 13] this system

extracts all the logical dependencies between attributes, or properties, of observed objects as recorded in a binary relation $R \subseteq T \times E$, where E denotes a collection of software events and T is a collection of execution traces. We let $\mathcal{L}$ denote the concept lattice generated by R. Each "concept" of $\mathcal{L}$ is a set of events that is closed with respect to the Galois closure of R. Let Z be a closed set. A subset $X \subseteq Z$ is said to be a generator of Z, if Z is the closure of X. Our interest is in minimal generators, that is those generating sets X for which no subset $X' \subset X$ generates Z. Each closed set, together with its generator(s) then determines a logical dependency. More specifically, if a subset $\{ac\}$ is a generator of the closed set $\{abcdef\}$ of events then, as shown in [12], $\{ac\}$ logically implies $\{abcdef\}$, as well as any subset of $\{abcdef\}$. The dependency, or implication, can be expressed in first-order notation as

$$(\forall t \in T)[a(t) \wedge c(t) \;\rightarrow\; a(t) \wedge b(t) \wedge c(t) \wedge d(t) \wedge e(t) \wedge f(t)] \tag{1}$$

where $a(t)$ means that a occurs somewhere in trace t, that is $(t, a) \in R$ or, loosely speaking, $a \in t$. Letting concatenation denote conjunction and suppressing the universal quantifier, we abbreviate (1) as simply

$$ac \;\rightarrow\; abcdef$$

Suppose bc is also a generator of $abcdef$, then we would have

$$bc \;\rightarrow\; abcdef$$

or

$$ac \vee bc \;\rightarrow\; abcdef$$

These expressions implicitly indicate that the closed set $\{abcdef\}$ has two generators.

In contrast to more common statistical data mining where item associations $often$ occur, the data mining performed by the DDDM system is deterministic because these implications $always$ occur — at least in all observations so far. 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.

We say that the DDDM approach is *discrete* because the universal quantification can only be over the finite domain T comprising the relation R.

The DDDM system constructs the concept lattice $\mathcal{L}$ incrementally in a manner that was first described by Godin and Missaoui in [4–6] and refined a bit more in [16, 17]. Incremental construction of the concept lattice facilitates the 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 [13].

3 Trace Data

To test this approach, the author used trace data generated by *JBoss* software. Jboss is 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 [19]. A small sample of this trace data from a single thread is shown below in Figure 1.

By an "event" in these traces we mean the invocation of a method. In other traces the notion of an event could be expanded, for example to include two variables becoming equal. But that fine level of granularity was not captured in the raw data given to us. Preprocessing consists of scanning each event in each of the 1,227 traces; and, if new, assigning it an identifying integer. The integers to

```
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. 1. A representative fragment of an event sequence

the left in Figure 1 are representative. This preprocessing has several benefits. First, it insures that the closed sets are extracted without using any embedded semantic information. Second, it permits us to represent a set of events as a set of integers, which we will see has definite *display* benefits. Third, because identifying integers are assigned in sequence as methods/events are scanned we have an interesting artifact in which related events 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 1 would be perceived by our DDDM software to be

$$\ldots 3\ 2\ 4\ 1\ 2\ 1\ 2\ 3\ 2\ 4\ 5\ 2\ 1\ 2\ 1\ 2\ 3\ \ldots \tag{2}$$

From now on we consider only discrete integer representations. We observe that many methods are repeated in definite patterns. A method for describing these repeating patterns can be found in [19]. However, our analysis is set based. All we can assert is that methods {1,2,3,4,5} all occur somewhere in this trace.

We analyzed 1,227 trace sequences consisting of 498,489 events of which 144 were distinct. The shortest trace consisted of no more than 6 events; the longest trace involved 1,405 events.

Given this data our DDDM software creates a concept lattice $\mathcal{L}$ with 1,805 closed concepts. No attempt to draw this lattice could be meaningful. We have the same problem as Lindig and Snelting. But later a fragment will be displayed in Figure 4. A smaller, synthetic example consisting of just 57 events in 8 thread sequences can be found in [11]. In that case a concept lattice of only 29 closed event sets, which is easy to comprehend, was generated.

4 Logical Implication

Section 2 gave us a quick sketch of how closed sets and their generators correspond to logical implication. In this section we expand on this theme.

Two representative concepts from the printed version of the concept lattice generated by our DDDM system are shown in Figure 2. The first "attribute" line enumerates the closed set of at-

```
concept #445:
        attribute-->    {2, 3, 13, 14, 17, 18, 19, 20, 21, 22, 23}
        object-->       << 1,100 >>
        generators-->   {{17}, {20}, {21}, {22}, {23}, {2, 18}, {3, 18},
                {13, 18}, {14, 18}, {2, 19}, {3, 19}, {13, 19}, {14, 19}}
        downpointers--> {446, 1744}

concept #448:
        attribute-->    {2, 3, 13, 14, 17, 18, 19, 20, 21, 22, 23,
        50, 53, 54, 55, 58, 59, 61}
        object-->       << 1,098 >>
        generators-->   {{50}, {53}, {54}, {55}, {58}, {61},
                {2, 59}, {3, 59}, {13, 59}, {14, 59}, {17, 59},
                {20, 59}, {21, 59}, {22, 59}, {23, 59}}
        downpointers--> {445, 1743}
```

Fig. 2. Two concepts from the printed output of the DDDM system.

tributes/events comprising the concept. The second "object" line enumerates the objects supporting the concept. (When we coded this print procedure we were analyzing closed sets of attributes, or properties, associated with objects.) The set of supporting object/trace sequences is a closed set in the object/trace domain. This dualism is well explained in [3]. To save space in this figure, we have replaced the actual enumeration of the 1,100 associated trace sequences of concept #445, and the 1,098 of concept #448 with their cardinalities.

The attribute/event "generators" of the concept are enumerated on the third line. Observe that events 17, 20, 21, 22, 23 are each, by themselves, generators of the closed set of #445. This means that if event 17 appears anywhere in the trace then all the events of the closed set must also occur in the trace. As explained in [13], because the closure operator is the Galois closure we can express the generation relationship as a logical expression $< generator > \rightarrow < closed\_set >$. Or more formally,

$$(\forall t \in T)[17(t) \rightarrow 2(t) \wedge 3(t) \wedge \ldots \wedge 22(t) \wedge 23(t)$$

where as before $17(t)$ is interpreted to mean that event 17 occurs in trace t, or $17 \in t$. Since the same can be said for event 20, or event 21, a more comprehensive logical implication based on concept #445 would be

$$(\forall t \in T)[17(t) \vee 20(t) \vee 21(t) \vee 22(t) \vee 23(t) \rightarrow 2(t) \wedge 3(t) \wedge \ \ldots \ \wedge 22(t) \wedge 23(t)$$

The presence of *any* of these generating events in a trace ensures the presence of all of the events in the closed set.

There are more entries in the generating set of concept #445. The set {2,18} is designated as a generator, that is, if event 2 *and* event 18 both occur in a trace, then all of the events of the closed set must be present as well. The corresponding logical expression is

$$(\forall t \in T)[2(t) \wedge 18(t) \rightarrow 2(t) \wedge 3(t) \wedge \ldots \wedge 22(t) \wedge 23(t)$$

But, if we want a single logical expression relating all generators of the concept to the closed set we get

$$(\forall t \in T)[17(t) \vee \ldots \vee 23(t) \vee (2(t) \wedge 18(t)) \vee \ldots \vee (3(t) \wedge 19(t)) \rightarrow 2(t) \wedge 3(t) \wedge \ldots \wedge 22(t) \wedge 23(t)$$

We can simplify the preceeding expression slightly by using concatenation to denote the logical *and* and eliding the universal quantification, to get

$$17 \vee 18 \vee \ldots \vee 23 \vee (2,18) \vee \ldots \vee (3,19) \rightarrow 2,3,13,14,17,18,19,20,21,22,23$$

But, this is still unwieldy. For example concept #272 has 1,435 disjunctive generators, some of which consist of as many as 4 conjunctive events.

One does not interpret these concept structures by eye! One does not try to draw an entire concept lattice.

For this paper, we will consider only closed event sets which have a singleton (event) generator. So we use a short procedure to traverse through the entire concept lattice and output the 66 closed concepts that have at least one singleton generator. A somewhat condensed version of its output is shown below. In particular, whenever possible we denote sets of 3 or more consecutive integers (event identifiers) by two dots. Here we have flagged with a * those concepts referenced elsewhere in this paper.

	concept	support size	singleton generators	closed set
	210	1,034	{1}	-> {1,2,3}
	837	1,160	{2}	-> {2}
	211	1,158	{3}	-> {2,3}
	250	977	{4}	-> {2,3,4}
	836	1,143	{5}	-> {2,5}
	273	3	{6}	-> {1..6}
	118	3	{7}	-> {1..5,7}
	40	3	{8}	-> {1..5,8}
	1789	2	{9}	-> {1..6,8,9}
	1	1	{10,11}	-> {1..11}
	1733	1,099	{12,15,16,24}	-> {2,3,12..24}
*	446	1,130	{13,14}	-> {2,3,13,14}
*	445	1,100	{17,20,21..23}	-> {2,3,13,14,17..23}
*	1744	1,103	{18,19}	-> {18,19}
	251	966	{25..32}	-> {2,3,5,12..32,56,57}
	389	938	{33}	-> {1..3,5,12..40,56,57}
	392	1,057	{34}	-> {1..3,5,12..32,34..38,56,57}
*	391	962	{35,36,37,38}	-> {2,3,5,12..32,35..38,56,57}
*	444	975	{39,40}	-> {2,3,13..23,39,40}
	443	977	{41,42,43,44}	-> {2,3,13,14,17..23,40..44}
	1735	863	{45}	-> {2,3,12..24,45}
	945	852	{46}	-> {2,3,5,12..24,46..51,53..55,58..64}
	458	1,077	{47..49,51,60,62}	-> {2,3,13..23,47..51,53..55,58..61}
*	448	1,098	{50,53..55,58,61}	-> {2,3,13,14,17..23,50,53..55,58,59,61}
*	455	960	{52}	-> {2,3,5,13..23,39..44,47..55,58..62}
	53	1,091	{56,57}	-> {2,3,5,12..24,56,57}
*	1743	1,101	{59}	-> {18,19,59}
*	449	865	{63,64}	-> {2,3,13..23,50,53..55,58,59,61,63,64}
*	74	167	{65}	-> {2,3,5,12..24,65}
*	375	28	{66}	-> {2,3,5,12..24,39,40,50,53..59,61,66,67,69..72}
*	150	96	{67}	-> {2,3,5,12..24,50,53..59,61,67,69..71}
*	1754	28	{68}	-> {2,3,5,12..24,50,53..59,61,65,68..72}
*	279	100	{69}	-> {2,3,5,12..24,50,53..59,61,69..71}
*	452	101	{70}	-> {2,3,5,12..24,50,53..59,61,70}
	581	102	{71}	-> {2,3,5,12..24,50,53..59,61,71}
	583	101	{72}	-> {2,3,5,12..24,50,53..59,61,71,72}
*	1745	65	{73,74,75}	-> {18,19,59,73..75}
	37	167	{76}	-> {76}
	1528	39	{77}	-> {1..3,5,12..24,39..44,47..51,53..62,77}
	966	252	{78}	-> {2,3,5,12..24,78,79}
	967	231	{79}	-> {2,3,5,12..24,79}
	358	157	{80}	-> {80}
	1181	61	{81}	-> {1..5,12..45,50,53..59,61,78,79,81}
	232	17	{82}	-> {82}
	1746	7	{83}	-> {18,19,59,73..75,83}
	725	3	{84,117,118}	-> {1..5,12..32,35..45,47..51,53..62,71..75,78,79,83,84,101,114..118}

```
   833        5   {85}                -> {76,80,85}
   575       62   {86,87,88}          -> {1..5,12..32,35..45,47..51,53..64,76,78..80,82,86..88}
*  272        1   {89}                -> {1..6,12..45,47..51,53..65,68..72,76,78..80,82,86..100}
   823        2   {90..100}           -> {2,3,5,12..24,45,47..51,53..65,68..72,76,79,80,90..100}
  1283        3   {95,96,99}          -> {2,3,5,1224,45,47..51,53..62,65,76,79,80,95,96,99}
   731        8   {101}               -> {1..5,12..32,35..40,47..51,53..62,78,79,101}
   439        1   {102,103,105..112} -> {2,3,13..23,39..44,47..55,59..64,70,76,102..112}
  1748        5   {104}               -> {104}
*  499        1   {113}               -> {1..5,12..32,35..64,73..76,78..80,82,86..88,113}
  1618        4   {114..116}          -> {1..3,5,12..24,39..44,47..51,53..62,71..75,83,114..116}
  1802        2   {119}               -> {2,3,5,12..24,45..51,53..64,76,80,119}
   764        1   {120}               -> {2,3,5,12..24,45..51,53..65,68..72,76,79,80,90..100,119,120}
  1274        1   {121}               -> {1..5,12..45,47..62,65,73..76,78..80,95,06,99,121}
  1508        1   {122}               -> {1..3,5,12..24,39..44,46..65,67..77,83,114..116,122}
  1676        3   {123..125}          -> {1..3,5,12..24,47..51,53..62,76,78,79,123..126}
  1804        4   {126}               -> {1..3,5,12..24,47..51,53..62,76,126}
  1742        3   {127..129}          -> {18,19,59,73..75,83,127..129}
  1747        4   {130,131}           -> {104,130,131}
* 1749        4   {132}               -> {132}
  1788        1   {133..144}          -> {1..6,8,9,12..64,76,80,82,85,133..144}
```

Fig. 3. Concepts with singleton generators.

In just the 66 concepts shown in Figure 3 there are interesting patterns emerging. Some concepts are supported by observations in hundreds of trace sequences; others may occur in only one trace. Some, such as #1749: $\{132\} \rightarrow \{132\}$, are obviously trivial. There are an unusual number of "consecutive" generating events. We will discuss this phenomena in some depth in Section 6. Finally, each of the 144 different event types found in these traces by itself generates a closed set. This is surprising; it is not true in other sets of trace data we have used. We do not know if this is somehow significant or just an accidental artifact.

A very small (and manageable) subset of the entire concept lattice $\mathcal{L}$ involving a few of these singly generated concepts is shown in Figure 4. The last "downpointer" line of Figure 2, which enumerates the concepts covered by any specific concept was used to hand draw this fragment. In

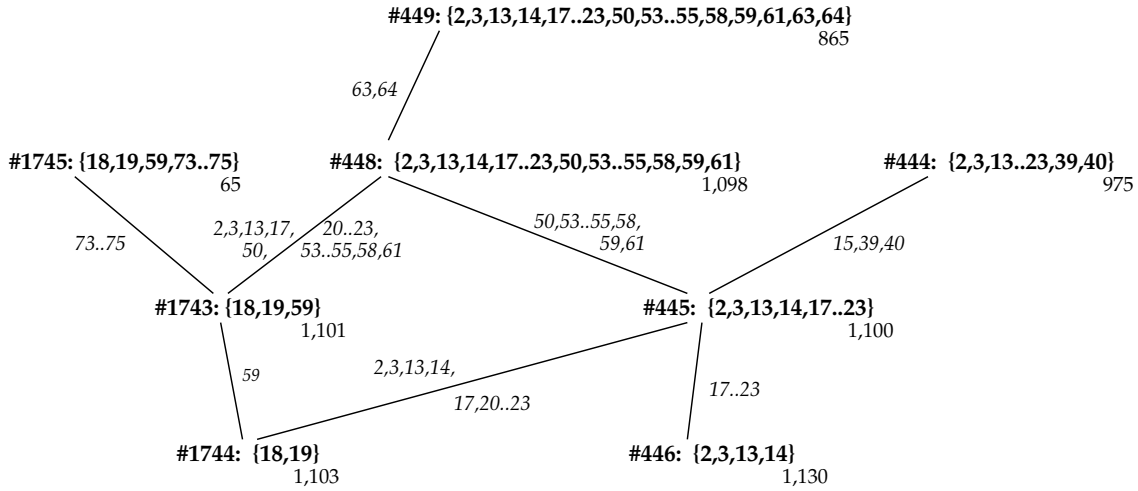

Fig. 4. A fragment of the 1,805 concepts comprising $\mathcal{L}$.

Figure 4, the events in the closed set have been enumerated. We have not indicated which events are singleton generators, but the concept numbers provide a link back to the listing of concepts with singleton generators in Figure 3. Below, and to the right, of each concept in Figure 4 we show the number of supporting traces. Because this is a dual representation to those concept lattices found in [3], these support numbers must be monotonically increasing as one descends through the lattice; just as the closed set cardinality must be monotonically decreasing.

If a closed set Z of events covers the closed sets $\{Y_i\}$ then beside each edge in Figure 4 we have shown the "closed set difference" $\Delta_{Z,Y_i} = Z - Y_i$, consisting of all events in Z but not in Y_i. Closed set differences are seldom indicated in concept lattice figures, but they are fundamental to our method of incrementally determining generating sets. Consider the collection of all generating sets, $Z.\gamma_k$ of Z. Let the closed set Z cover the closed sets Y_i in $\mathcal{L}$. As first proven in [7], $X \subseteq Z$ is a generator of Z if and only if $(\forall i) X \cap Z - Y_i \neq \emptyset$, that is at least one event of every difference set, Δ_{Z,Y_i} is represented in X. The set X is a *minimal* generator if no proper subset of X is a generator. The reader is encouraged to verify this theorem using the concepts of Figure 2 and Figure 4. In particular, why is {17} a generator of concept #445, but not {18}? Why is {2, 19} a generator?

5 Causal Dependency

Let $S = \{s_k\}$ be a collection of ordered sequences of integers, events, or whatever.[1] By $i <_k j$ we mean that event i preceded event j in sequence k. If the s_k can be ordered sequences with repetition, then by $i <_k j$ we mean that an instance of i precedes the $first$ instance of j. We say that i *dominates* j in S if $i <_k j$, in all $s_k \in S$ where both occur. Because of the universal quantifier in the definition of domination it is much easier, given actual sequences, to discover pairs i, j where i *cannot* dominate j because in some sequence $j <_k i$.

As our DDDM software reads each trace sequence, t_k, to incrementally construct the lattice of closed event sets, it also checks for *non-domination*. If an event j is found to precede the first occurrence of event i in some trace, t_k, then i cannot dominate j and `cant_dominate[i, j]` is set to true (1). A portion of this 144×144 array, `cant_dominate`, is shown in Figure 5. It plays a key

		61									71										81			
65	···	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0
66	···	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	1	0	1	1	1	0	1	1
67	···	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
68	···	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	0	1	1	1	0	1	0
69	···	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1
70	···	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1

Fig. 5. A small portion of the `cant_dominate` array.

role in our post processing to extract causal dependencies.

Logical implication is not equivalent to causal dependency. The accepted concept of "causality" involves time, whereas logic does not. If an event, such as 65 in concept #74, is the generator/precedent then we expect that this concept must precede the occurrence of the events {2, 3, 5, 12,

[1] A more precise definition of a sequence s_k can be framed as a function $s_k : [1, \ldots, m] \rightarrow [1, \ldots, n]$ where m denotes the length of the sequence, or trace, and n denotes the number of distinct events.

..., 24} of the closed consequent in order to say that 65 "causes" these events as a consequence. But inspection of each of the 167 trace sequences supporting concept #74 reveals instances where each of these other operator events precede the first occurrence of event 65. Event 65 does not dominate any of these other events.

Only if an event j never precedes the first occurrence of event i in any trace can we assume that a logical implication $i \rightarrow j$ can be rewritten as a plausible causal dependency, which we denote by $i \Rightarrow j$. We observe that if $i \rightarrow j$, then in every trace sequence where i occurs, j must also occur; and if j never occurs before i in *any* trace, then an instance of i must occur before the first instance of j in *every* trace where i occurs. It is important to note that $i \Rightarrow j$ is only a plausible causal dependency. We cannot establish that event i actually causes event j to occur. We can only establish that they satisfy the necessary conditions for "causality". We will discuss this further in Section 6.

We now examine concepts #375, #150, #1754, #279 and #452 of Figure 3 with singleton generators {66}, {67}, {68}, {69}, {70} respectively. Our question is whether any of these correspond to functional dependencies. Consider, for example, #375: {66} → {2, 3, 5, 12, ..., 24, 50, 53, ..., 59, 61, 67, 69, 70, 71}. Although not illustrated in Figure 5, all events 2 through 59 of the closed set precede 66 in at at least one trace. Referring to Figure 5, we see that events 61, 71 and 72 also precede 66 in at least one trace, while 67, 69 and 70 do not. So a plausible functional dependency is

$$\{66\} \Rightarrow \{67, 69, 70\}$$

Using a similar analysis we see that

$$\{68\} \Rightarrow \{67, 70, 71, 72\}$$
$$\{69\} \Rightarrow \{70\}$$

while the singleton events {65}, {67}, {70} as shown by the evidence of Figure 5 cannot functionally determine any other events in the closed sets that they logically imply.

In Figure 6, we summarize all of the causal dependencies that can be inferred from the logical implications of Figure 3 and the `cant_dominate` array.

6 Observations and Conclusions

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 $R \subseteq T \times E$; (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/or exit, from modules, methods or other bodies of code as in Ball [1] and in 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, 2, 10].

Michael Ernst, in particular, has been a leader in identifying likely invariants from dynamic trace data [2, 10]. 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

```
        support
concept size           causal dependencies (possible)

1733    1,099   {12} => {13...24}
 445    1,100   {17} => {22,23}
                {20} => {21...23}
                {21} => {22,23}
 251      966   {25} => {26} => {27} => {28} => {29}
                {28} => {30} => {31} => {32}          (Note: {29} /=> {30...32})
 391      962   {35} => {36} => {37} => {38}
 444      975   {39} => {40}
 443      977   {41} => {42} => {43} => {44}
 945      852   {46} => {47,48,49,60,62}
 458    1,077   {47} => {48} => {49} => {51} => {60} => {62}
 448    1,098   {50} => {53} => {54} => {55} => {58} => {61}
  53    1,091   {56} => {57}
 449      865   {63} => {64}
 375       28   {66} => {67,69,70}
1754       28   {68} => {69,70,71,72}
 279      100   {69} => {70}
1745       65   {73} => {74} => {75}
 966      252   {78} => {79}
 725        3   {84} => {117} => {118}
 575       62   {86} => {87} => {88} => {25...44,45,63}
 272        1   {89} => {33,34,90...100)
 823        2   {90} => {91}=>{92}=>{93}=>{94}=>{97}=>{98}=>{100}
1283        3   {95} => {96} => {99} => {65}
 439        1   {102}=> {103}=>{106}=>{107}=>{108}=>{109}=>{110}=>{111}=>112}
 499        1   {113}=> {46...63,78,79}
1618        4   {114}=> {115} => {116} => {83}
1802        2   {119}=> {23...51,55...58,60...64}
 764        1   {120}=> {46...55,58...64,68...72,91...100}
1274        1   {121}=> {35...45,52,65,95...99}
1508        1   {122}=> {83}
1676        3   {123}=> {124} => {125} => {1,2,3,5,12...24,47...51,53...62,78,79}
1804        4   {126}=> {1,2,3,5,12...24,47...51,53...58,60,61,62)
1742        3   {127}=> {18,73...75,83,128}
                {128}=> {59,129}
                {129}=> {19}
1747        4   {130}=> {131}
1788        1   {133}=> {134}=>{135}=>{136}=>{137}=>{138}=>{139}=>{140}=>{141}=>{142}
                {142}=> {1...6,8,9,12...58,60...64,80,82,119,143,144}
                {143}=> {1...6,8,9,12...58,60...64,80,144}
                {144}=> {1...6,8,9,12...58,60...64}
```

Fig. 6. Possible causal dependencies.

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 6 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 others do not. For example, it is easy to show in a first-order logic that 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. One can easily construct examples where $a \Rightarrow x$ and $b \Rightarrow y$, yet $ab \not\Rightarrow xy$ because a does not dominate y or b does not dominate x.

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. We consider the closed set #391 for which events 35, 36, 37, 38 are all generators. Since each generates the same set, all are logically equivalent. But, they are not causally

equivalent. It would appear in this case that the *JBoss* software employs a stack architecture in which procedure 35 invokes 36 which invokes 37 and finally 38.

If x precedes y in at least one trace t_k we say $x < y$. Readily, $<$ is only a pre-order, since it is transitive but not necessarily antisymmetric. If $x < y$ then y cannot causally determine x. By creating a `cant_dominate` precedence relation such as Figure 5 in parallel with the concept lattice, we 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 we wonder if a precedence relation such as Figure 5 can do more. Why not use an anti-symmetric sub-relation to directly indicate causal dependencies? For example, we see in Figure 5 that $66 < 81$ while $81 \not< 66$. Thus we know that event 66 precedes 81 in at least one trace, and that 81 never precedes 66. 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 $66 \Rightarrow 81$ is that whenever event 66 occurs then 81 must also always ensue. But, in this set of events $66 \not\rightarrow 81$. This second necessary condition whose logical expression is

$$(\forall t \in T)[b(t) \rightarrow [b(t) < a(t)]] \tag{3}$$

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 make use of Galois closure, which is the basis of formal concept analysis. Only by adopting a formal concept methodology can we derive a universal expression such as (3). 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 $a \Rightarrow x$ in the set T of trace data, we cannot literally say that the event a "causes" the event x 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. Nevertheless, our work can provide a mechanism for discovering "likely" dependencies.

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. 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.
3. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis - Mathematical Foundations.* Springer Verlag, Heidelberg, 1999.
4. 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.

5. Robert Godin and Rokia Missaoui. An Incremental Concept Formation Approach for Learning from Databases. In *Theoretical Comp. Sci.*, volume 133, pages 387–419, 1994.
6. Robert Godin, Rokia Missaoui, and Hassan Alaoui. Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices. *Computational Intelligence*, 11(2):246–267, 1995.
7. Robert E. Jamison and John L. Pfaltz. Closure Spaces that are not Uniquely Generated. In *Ordinal and Symbolic Data Analysis, OSDA 2000*, Brussels, Belgium, July 2000. Discrete Appl. Math., (to appear), 2004.
8. 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.
9. Oystein Ore. Galois Connexions. *Trans. of AMS*, 55:493–513, 1944.
10. 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.
11. John L. Pfaltz. Using Concept Lattices to Uncover Causal Dependencies in Software. In B. Ganter and L. Kwuida, editors, *Proc. Int. Conf. on Formal Concept Analysis, Springer LNAI #3874*, pages 233–247, Dresden, Feb. 2006.
12. 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.
13. 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.
14. Michael Siff and Thomas Reps. Identifying Modules via Concept Analysis. In *Intn'l Conf. on Software Maintenance*, pages 170–179, Bari, Italy, Oct. 1997.
15. 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.
16. 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.
17. 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.
18. Rudolf Wille. Restructuring Lattice Theory: An approach based on hierarchies of concepts. In Ivan Rival, editor, *Ordered Sets*, pages 445–470. Reidel, 1982.
19. 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.