Accurate OO Static Analysis of Java Libraries to Predict Potential Deadlocking Call Stacks

1. Type level Abstract Interpretation of Method Body

Byte-code in the class file is meant for a virtual stack based machine that is implemented by the JVM using a *locals array*, and an *operand stack* that it maintains in each activation record corresponding to every method invocation on the call stack of every executing thread of the application.

The local state of the executing method body is maintained in its locals array[1]. For an instance method, the slot zero of this array is initialized with a reference to the target object of the invocation (the `this` pointer). Its next few slots of are initialized using the invocation arguments passed by the caller method. The operand stack initially is empty, and execution always starts from the first instruction in the byte code sequence. After having thus initialized execution of the invoked method, the values of accessible variables and objects drive the manner in which execution proceeds, contributing to the evolution of the state of the JVM.

Our abstract interpretation also maintains and uses the locals array, and the operand stack discussed above, but (unlike the JVM) we populate their entries with *types* of the variables/objects rather than their values. Which is why we term it *type level* abstract interpretation. In this section we shall look, in detail, at abstract interpretation of the method body with a view to identify the type of the object locked at its every `monitorenter` code point. The byte-code instructions `monitor enter/exit`, used to lock/unlock an object, retrieve the reference of the object from the top of operand stack.

Consider, for instance, there is a method with signature: `Fscn.mn:(P1, P1)V`, where `Fscn` is the fully specified class name that defines method `mn`, which takes two formal parameters, of types `P1` and `P2`, respectively, returning `void`. To simplify the discussion in this section we assume that we are analysing the invocation of `mn` on a target object whose actual type is indeed `Fscn`, and the types of its first and second arguments are also indeed `P1`, and `P2`, respectively. Furthermore, since our analysis is static time, we don't model the heap, nor actual objects, nor values.

Consequently, we begin abstract interpretation by initializing the locals array using the (encodings of) *type* names of the objects as follows: its slot zero holds the name of the containing class, its slot one will hold (encoding of) the type name of its first parameter, and so on. In the same spirit, below, in this subsection we *indicate* the semantics attributed to abstract interpretation of the various Java byte-code instructions. Just like the way the JVM Spec[**?**] lays down the specification of the semantics associated with the execution of same list of Java byte-code instructions, albeit more formally. We start with identifying certain essential inputs, computed state components and outputs:

- Inputs `offset[]` and `code[]` are two arrays (of equal length), that together store the text rendering of the byte-code corresponding to the method that is to be abstract interpreted. `offset[i]` identifies the index into the byte-code array of the i^{th} instruction, and `code[i]` stores the text rendering of it, including the significant comment as generated by `javap`.
- An input `monitors` is a map as discussed in Section 2.1. Since the purpose of abstract interpretation is to discover the type of the object locked by the execution of `monitorenter`, we take note of the *type* at the top of the operand stack *before* interpreting the `monitorenter` instruction.
- The variables `tA`, and `tS` are used to represent the locals (type) array, and the operand (type) stack, respectively. They are initialized as discussed earlier. The (input) length of `tA` is (earlier) read from the class file. The actual types stored at the various indices of the `tA` keep getting updated as abstract interpretation progresses. So also the composition and the depth of `tS`.
- We categorize the various Java types into three categories: ‘c1’ (short for category 1, as the VM-Spec[**?**] describes them) represents the one word basic types of Java that comprise of `int`, `short`, `byte`, `char`, and `float`; ‘c2’ (short for category 2) represents the two word basic types of Java that comprise of `long` & `double`; lastly the

[1] Based upon the parameter count of the method definition and the local variables declared in the method body, the compiler would have computed the required (maximum) size for this array, and stored its value, as a method body preamble, in the class file as advice that the JVM uses when conducting its execution.

reference types that are represented by their type name[2]. For example to represent `int[][]`, the java compiler would encode it as '[[I' but our abstract interpretation records it as '[[c1'.

- The VM-Spec discusses the layout of its operand stack, and the locals array in terms of 'words'. In the same spirit, we maintain our `tA`, and `tS` in terms of 'words'. Therefore abstract-type 'c2' occupies two contiguous slots whenever `tA` or `tS` are read from or written into. The JVM incorporates a byte-code verification step that ascertains (among other things) that the byte-code for a loaded class/method doesn't attempt mischief like interpreting two contiguous words of type 'c1' as an object of type 'c2'. While our abstract interpretation largely works under the assumption that the input being analyzed is 'well formed', we nevertheless assert (at every available opportunity) that the (abstract) types of operands on the stack do indeed match the types anticipated by the specific instruction, so as to ascertain the 'well-behavedness' of our interpretation of the byte-code.

  For instance when faced with the need to interpret an 'iadd' instruction, we ascertain that the top two slots of `tS` indeed contain 'c1', each, and we pop both of them to symbolize their consumption. Subsequently we push 'c1' onto it, to symbolize the production of the result of its computation. Whereas, when the instruction to be interpreted is 'dadd', we ascertain that the top 4 slots of `tS` contain 'c2', each. We 'consume' all of them, and then again push 'c2' onto it twice to symbolize the production of a double value from the computation. A more elaborate list of instruction types, actual examples, and our actions involved in their interpretation are listed in Table 1, columns 1, 2 ,and 3, respectively.

- Consider a control transfer instruction like `iflt` that compares an integer value from the top of the operand stack with zero, and transfers control to the target byte-code index if the value is negative. There are therefore potentially two ways the control could flow after executing such an instruction: either it could continue with the next instruction in the sequence, or it could transfer to the target address. In either case, the state of the `tS`, and the `tA` would be just as the current instruction leaves it, for both code points.

  Our Abstract interpretation maintains a 3-tupple: `execStartStates` $\equiv<$ `execStartOffset`, `StacksAtExecStart`, `arraysAtExecStart`$>$. To interpret `iflt` we firstly ascertain that there is 'c1' on the top of `tS` before consuming it, then we insert/update an entry into this 3-tupple with the target address as the value for `execStartOffset`, and we store a clone of the current state of the `tS`, and `tA` objects into the other two fields of the entry. Having done this we continue with interpretation of the sequentially next instruction.

- Our implementation therefore uses two methods to accomplish the task of abstract interpretation of an entire method body. The higher level method (`processMethodBody`) helps discover, and ensures interpretation of, all the various subsequences of the method body that *could* execute entirely if control were to enter it through its first instruction. (These subsequences are therefore not quite like the basic blocks in code compiled for RISC/CISC processors, where the semantics is that all its instructions *would* execute *iff* its first one did). The start points of these subsequences are stored in the form of 3-tupples discussed above, and their union covers the entire method body (unless it contains dead or unreachable code). The other method (called `interpret`) implements the actual abstract interpretation of a single instruction at a time, given all the above described state, and advices its caller `processMethodBody` regarding the end point of subsequences. The column 3 in Table 1 contains actual java source fragments from the `interpret` method.

- `processMethodBody` invokes `interpret` which actually updates the contents of `tS`, and `tA`, to reflect the abstract interpretation of the given instruction. `interpret` is invoked in a loop, every time handing it the sequentially next instruction to interpret. However this loop execution continues only so far as `interpret` continues to return `true`. For byte-code instructions like *[adfil]return* it returns `false`, to indicate that the execution trail terminates there, and the sequentially next instruction can not be analyzed assuming the `tS`-state and the `tA`-state as the current instruction would leave it. Similarly `interpret` returns `false` for *goto(_w), jsr(_w), ret, lookupswitch, tableswitch & return*. For all other control transfer instructions, `interpret` returns `true`.

- The analysis of the method body therefore happens in parts, starting from either the beginning, or from the target of some control transfer instruction, and going on until `interpret` returns `false`. Every time this inner loop terminates, the next target of some control transfer instruction is read from the `execStartOffset`, and the `tS` and `tA` are accordingly initialized. The loop so initialized continues the analysis of the corresponding subsequence of the method body.

- Instructions *jsr / jsr_w* push the offset of the (sequentially) following instruction on the operand stack before transferring control to the address available as its operand. The *ret* instruction transfers control *back* to the return address stored in the local variables array at the index identified by its operand. Our abstract interpretation of these instructions uses the `tS`, and the `tA`, just as the JVM would use its operand stack and its locals array:

[2] Where the type name happens to be *java.lang.Class*, and our analysis is further aware that the specific object happens to be representing java.lang.String, then we record the type as *java.lang.String.class*

we store/retrieve the actual value of the *return address*. Note that in this (exceptional) case the *value* is a piece of static information that is the same for all executions of this code.

The `interpret` method implements the semantics as indicated in Table 1. Given a specific instruction, `interpret` asserts that `tS` and `tA` are composed as the specific instruction would expect. Further, it updates them according to the semantics of the instruction, and also updates the 3-tupple in case the instruction so demands. Finally it returns either `true` to indicate that the updated `tS` and `tA` are appropriate initial states for the interpretation of the sequentially following instruction or `false` otherwise.

Table 2 lists the details for the monitor instructions. `processMethodBody`, an instance method of the `JavaBlob` class which implements a node of the call-graph is invoked after reading the method body into its field attributes: `offset[]`, and `code[]`. To enable handling of arbitrary nesting of `synchronized` statement blocks, this method maintains, locally, a stack (`oS`) of call-graph nodes. Upon entry, it initializes `oS` by pushing `this` onto it. Subsequently, on encountering `monitorenter` it creates a new node (per Section 2) with the appropriate name and pushes it onto `oS`. Later, on encountering the (*last matching* `monitorexit` as confirmed by the `monitors` map, it 'closes' processing of the corresponding node of the call-graph, by popping it off `oS` (while adding an directed edge from `oS.peek()` to `os.pop()`). Any method invocations (as per Table 1) encountered in the meanwhile, (*i.e.* in between the `monitorenter` and its *last matching* `monitorexit`), will get recorded using directed edges originating from `oS.peek()`.

The rest of the byte-code instruction categories are in Table 3, which is discussed in Section 3. Together the three tables cover the entire set of categories of Java byte-code instructions.

1.1 Run time Types driven Abstract Interpretation

Table 3 lists the rest of the categories of byte-code instructions including some examples of how their abstract interpretation is conducted. We choose the manner of abstract interpretation of these instruction sets so that the run time types of objects get fed into the computation of call-graph edges, and thereby the *tlog*. The following list of general design guidelines fetch the specific action sequences in column-3 for the corresponding byte-code instructions in the column-2 of the Table:

- Recording the actual type of object instantiated by `new`.
- Recording the precise number of dimensions, and the element type instantiated through `newarray`, `anewarray`, and `multianewarray`.
- Using the actual type names of the arguments passed to method invocations through a member of the 'invoke' family of instructions. Similarly, recording the actual type of the object that is target of the method invocation.
- Storing class and interface hierarchies for the entire library and using these hierarchy chains to identify exactly which super class, or subclass method will get invoked given the run time types recorded above, and in view of the specific semantics of the particular member of the invoke family of instructions.

The above list is complemented by the following two modifications in the structure of the call-graph, and the structure of their nodes. Together these augmentations to the infrastructure supporting the analysis enables us to predict the dynamic dispatch of java method invocations, in a static analysis framework, reasonably well.

- We add another collection of nodes into the call-graph called the `varients`, that contain nodes corresponding to every unique two-tuple of the form <*i-sign*, *b-spec*>, where *i-sign* is an invocation signature generated during abstract interpretation of some invoke instruction within some method body, and *b-spec* is identified as the method signature whose method body definition is to be used to generate the *interpretational varient behavior* represented by the node <*i-sign*, *b-spec*>. The node name of the corresponding node is a concatenation of: *i-sign*, '-:varientOf:-', and *b-spec*.
- Members of the `varients` set have a non null attribute called `bSpec`, that stores the reference to the node that supplies the method body definition, as described above.

1.2 Accurate Call graph for OO s/w

Our analysis is implemented by a single tool, currently, that parses the various *jars* from the *jre/lib/* subdirectory (including, but not limited to the *rt.jar*) produced from a successful build of the JDK. Our implementation of Step 1, below, comprises of two concurrent threads (organized in the producer consumer model): one that prepares the input, and another that consumes the so prepared input in a single pass to create the data structures for analysis.

The thread that prepares the input starts with a `'jar -tvf'` command to fetch the list of class files that are available in the *jars*. It then uses `'javap -private -c -verbose'` to have each of the class files rendered into text. This text rendering of the class file from the jars is then consumed by the other thread to create the data structures for subsequent analysis. The rest of the analysis, after Step 1, is single threaded. The 5 top level algorithmic steps of the analysis are as follows:

- **Step 1**: Read the entire source available in the input, in a single pass, while storing the essential information into the following set of attributes that help form the call-graph:

Instruction Description, and specification	Actual example from text rendered byte-code	Semantics
not an operation	nop	–
Unary Arithmetic: [ilfd]neg	ineg	assert (tS.peek()==c1)
	dneg	assert (tS.pop()==c2); assert (tS.peek()==c2); tS.push(c2);
Load a local onto stack [adfil]load_[0123], [adfil]load	aload_3	tS.push(tA[3]);
	dload 7	assert (tA[7]==c2); tS.push(c2); assert (tA[8]==c2); tS.push(c2);
Store top of stack into a local [adfil]store_[0123], [adfil]store	istore_2	assert ((tA[2]=tS.pop())==c1);
	dstore 6	assert ((tA[6]=tS.pop())==c2); assert ((tA[7]=tS.pop())==c2);
Load constant onto stack: [bs]ipush, iconst_m1, fconst_[012], aconst_null, [dl]const_[01], iconst_[012345]	sipush 931	tS.push(c1);
	dconst_0	tS.push(c2); tS.push(c2);
	aconst_null	tS.push(“Ljava/lang/Object;”);
Numeric Conversions: [ilfd]2[bsilfd]	i2l	assert (tS.pop()==c1); tS.push(c2); tS.push(c2);
	d2f	assert (tS.pop()==c2); assert (tS.pop()==c2); tS.push(c1)
Binary Arithmetic: [ilfd]((add)|(sub)|(mul)|(div)|(rem)), Bitwise: [il]((or)|(xor)|(and)), Shift: [il]sh[lr], [il]ushr Comparison: [fd]cmp[gl], lcmp	fadd	assert (tS.pop()==c1); assert (tS.peek()==c1);
	ixor	assert (tS.pop()==c1); assert (tS.peek()==c1);
	lshr	assert (tS.pop()==c1); assert (tS.pop()==c2); assert (tS.peek()==c2); tS.push(c2);
	fcmpg	assert (tS.pop()==c1); assert (tS.peek()==c1);
Load array component onto stack: [bcsilfda]aload	caload	assert (tS.pop()==c1); String aN = tS.pop(); tS.push(‘c1’); assert (aN==‘[...’) || aN==‘Ljava/lang/Object;’);
Store top of stack into array component: [bcsilfda]astore	iastore	assert (tS.pop()==c1); assert (tS.pop()==c1); String aN = tS.pop(); assert (aN==‘[...’) || aN==‘Ljava/lang/Object;’);
Get length of array: arraylength	arraylength	String aN = tS.pop(); tS.push (c1); assert (aN==‘[...’) || aN==‘Ljava/lang/Object;’);
Stack operations: ((pop)|(dup))2?, dup_x[12], dup2_x[12], swap	dup_x1	String ts=tS.pop(), tsn=tS.pop(); tS.push (ts); assert (ts!=c2 && tsn!=c2); tS.push(tsn); tS.push(ts);
Conditional branch: if(non)?null, if((ne)|(eq)|(l[te])|(g[te])), if_icmp((ne)|(eq)|(l[te])|(g[te])), if_acmp((ne)|(eq))	ifle 1420	assert (tS.pop()==c1); execStartStates.add (1420, tS.clone(), tA.clone());
	if_acmpeq 118	String ts=tS.pop(), tsn=tS.pop(); assert (ts!=c1 && ts!=c2 && tsn!=c1 && tsn!=c2); execStartStates.add (118, tS.clone(), tA.clone());
Compound conditional branch: tableswitch, lookupswitch	lookupswitch{ //1 ... ... 28: 24; default: 29 }	execStartStates.add (24, tS.clone(), tA.clone()); execStartStates.add (29, tS.clone(), tA.clone());
Unconditional branch: ret, goto(_w)?, jsr(_w)?	jsr 303	tS.push (nextInstrOffset); execStartStates.add (303, tS.clone(), tA.clone());
Rest of the instructions: athrow, wide, iinc, ldc_w, ldc2(_w)?	ldc_w #220;... ... //class java/net/Socket	String cObj = ‘java/net/Socket’; tS.push (‘L’ + cObj +‘.class;’);

Table 1. Abstract interpretation for some Java byte-code instruction categories

Actual example from text rendering of byte-code	Semantics
monitorenter	assert ((ts=tS.pop())!=c1 && ts!=c2); String lt = tS.pop(), nm = oS.peek().getName() +‘#’ + lt + ‘#’ + counter++; oS.push (new JavaBlob (name=nm, lockType=lt));
monitorexit	assert ((ts=tS.pop())!=c1 && ts!=c2); if (monitors.valueSet().contains (currInstrOffset)) { assert ((tmp=oS.pop()).getLockType() == tS); oS.peek().noteCallTo (tmp); }

Table 2. Abstract interpretation for monitor enter & exit instructions

Instruction Description, and specification	Actual example from text rendering of byte-code	Semantics
Create an instance: new	new #200; //class java/lang/String	String tN = “java/lang/String”; //read type name tS.push(‘L’ + tN + ‘;’); // push its signature
Create an array: newarray, anewarray, multianewarray	newarray char	assert (tS.pop()==c1); //consume the array size tS.push(‘[’ + c1); // ‘char’ encodes to c1
Method Invocation: invokespecial, invokestatic invokeinterface, invokevirtual	invokestatic #101; ... ...//Method nextIndex:(II)I	assert (tS.pop()==c1); assert (tS.pop()==c1); //use 2 args oS.peek().noteCallTo (classNm + ‘.’ + ‘nextIndex:(II)I’); tS.push (c1); //make the return value available
Access Class or Instance Field: ((get)|(put))((field)|(static))	getfield #356; //Field value:[C	String tN = “[c1”; //*rhs* of the ‘:’ in the tS.push(tN); // significant comment
Check Type properties: instanceof, checkcast	instanceof #51; //class java/util/Map	String ts=tS.pop(); tS.push(c1); assert (ts!=c1 && ts!=c2);
Various returns: ret, return [ilfda]return	ret 4	execStartStates.add (tA[4], tS.clone(), tA.clone());
	dreturn	assert (tS.pop()==c2); assert (tS.pop()==c2);

Table 3. Semantics of abstract interpretation for Java byte-code instructions

- `definedCode`: A mapping from a method signature (or some name mangled form of it) to the node that represents the corresponding method body (or a `synchronized` statement block). Any piece of code that is part of the input is represented by some node on this map. However, this could contain method signatures whose behaviors are not defined in the input.
- `nativeMethods`: The set of method signatures of native methods.
- `abstractOrInterfaceMethods`: The set of method signatures of abstract or interface methods.
- `classes`: A map from the name of a class to a `ClassInfo` object that stores the name of its parent class, and the list of all the interfaces it implements.
- `interfaces`: A map from the name of an interface to a the list of all the interfaces it extends.
- `sCode`: A map that starts as a subset of `definedCode`. During the Step 1, while input is still being read, this map is updated to include such entries from the `definedCode` map that correspond to code tagged `synchronized` in the source.

The value objects in `definedCode`, and `sCode` maps above are nodes of the call-graph, and directed edges amongst them form the edge set. Objects of class `JavaBlob` are used to represent the nodes. Its following other attributes are populated in Step 1:

- `nodeName`: See Section 2.
- `lockType`: If the node represents `synchronized` code, this attribute records the type of the object locked during execution. However for a node representing a `static synchronized` method, it records the containing class name postfixed with a ‘.class’.
- `isPublic`: `true` if node represents a `public` method.
- `isStstic`: `true` if node represents a `static` method.
- `accuracy`: This attribute stores the level of analysis conducted on the method body represented by this node. The three possible values for it are: `Null`, `CompileTimeType`, and `RunTimeType`. For a node whose corresponding method signature was encountered as a compiler resolved called method, but whose corresponding method definition is not seen, this is `Null`. For nodes corresponding to methods defined in the input, this attribute stores the `CompileTimeType` value.
- `locals`: This integer value is the max size of the locals array estimated by the compiler.
- `<offset[], code[]>`: The method body.
- `monitors`: See section 2.1.
- `calledCode`: This attribute is a map. For nodes in `definedCode` whose `accuracy` attribute remains set to `Null`, this map remains un populated. When nodes (whose method bodies) are analyzed at `CompileTimeType` level, for every *invoke* family of byte-codes that is encountered therein, an entry is added into this map using the compiler resolved method signature as its key, and a reference to the corresponding node from the `definedCode` map as the value.
- `callingCode`: This is a map whose purpose is to store references of all nodes whose `calledCode` refers this node. Whenever the `calledCode` attribute value is updated, this attribute for the appropriate nodes also needs to be set right.

Concurrently, while creating the call-graph described above, we also populate the `tlog`. This is a collection of objects (of type `OTlock`), that represent locks. Every unique value assigned to any `lockType` attribute of a node on the `sCode` subset of the call-graph, is represented here by an object. Every time a node is added into `sCode`, we ensure that the type name stored in its `lockType` attribute is represented in this `tlog`-graph. The following attributes of the object are populated in Step 1.

- `lockName`: this attribute derives its value from the value of the `lockType` attribute of the call-graph node. Since this is used as the key in the map that stores the `tlog`, it must be non null.
- `lContenders`: This attribute stores the collection of call-graph nodes that, each, when invoked, shall require to lock some object represented by this lock before proceeding to compute the code it represents.

- **Step 2**: Inherit parents' functionality: In this step we iterate over all nodes from `definedCode` whose `accuracy` attribute is initialized to `Null` level, investigating, given such a node (say, `m`), in view of the class hierarchy, whether any super class method (say `mb`), defines the behavior which can be inherited by `m`. If so, then we set: `m.bSpec = mb`, and use the method body of `mb` to upgrade the analysis level of `m` to `CompileTimeType`. Failing which, on the other hand, if m turns out to be some inherited super class abstract method declaration (say `ma`), then we flush m from `definedCode`, re-adjusting edges directed into m, to instead point into `ma`.
- **Step 3**: Identify call stacks acquiring 2 locks: In this step we iterate over nodes from the `sCode` map, and for each node `s` therein, we do the following:
  1. Identify a list of public *concrete* potential invokers of `s`. We use the `callingCode` attribute of nodes for this reverse walk along a potential call stack. A method signature is *concrete* if its target object type, and all the types of its parameters are types that can be instantiated. Abstract types, and interface types cannot be instantiated. The result of this computation is stored into `s.concPublicInvokers`. In order to avoid looping forever around cycles in the call-graph, we use a `FanOut` counter (initialized to 7, say) that counts down every time we 'walk' past a node. A path is explored only while the counter value is positive.
  2. We explore all call paths starting from every member of `s.concPublicInvokers`. While doing so, we always upgrade the `accuracy` of every node on every path to the `RunTimeType` level of analysis. This ensures that the information recorded in their `calledCode` attributes is *accurate* before we use it to get further along the path. While 'walk'ing along these (potential) call paths, we record the names of nodes passed on the path, until some other lock is acquired along a path, say `t.lockType`. Ensuring that the first `lockType` acquired along the path is indeed `s.lockType` avoids our analysis from reporting cycles that are protectd by some *fat lock*. As soon as such a node `t` is reached, we record the entire call path starting from the member of `s.concPublicInvokers` that started the path, and until the node `t`, into an attribute `s.sEdges` (of type Set), and back track.

  When nodes get upgraded to `RunTimeType` level in step 3.2, it can be the case that new invocation signatures are encountered for which correspondingly new nodes need to be created in the call-graph. These nodes are stored in the `varients` map. Further, if any of them happen to be inheriting their behavior specifications from `synchronized` method bodies, then they are additionally enlisted into the `sCode` map.
- **Step 4**: Compute `tlogEdges`: This attribute of `OTlock` (the type of `tlog` nodes), stores the set of directed edges that each represents a path in the call-graph. Some such edge (a –> b) would therefore correspond to a call path starting at some (concrete public invoker of some) member of `a.lContenders`, and represents nested method invocations of method bodies that eventually invoke some method that is a member of `b.lContenders`. More specifically, `a.tlogEdges` is a map: the name (`b`) of the *other* `tlog`-node is held as the key, and the value is the set of paths that all represent different call paths that each, eventually, request a lock on an object of type `b`, having started off by firstly acquiring some lock of type `a`. A path is stored as an ordered hyphenated concatenation of the node names of the nesting sequence of method bodies.
- **Step 5**: Detect and print cycles in the `tlog`: Cycles in the `tlog` reflect the fact that the programmers of the input source have not followed any resource ordering when designing the library implementation. As the final output of our analysis, we print starting from 2-lock cycles upto n-lock cycles, where n is configurable. Given a 2-cycle, there can be various ways to realize each of the two different thread stacks that can deadlock the JVM. All these options are co-located in the output format, so as to ease the task of a library programmer interested in using this analysis to improve the library design / implementation. In general, an n-cycle consists of (n+1) components. An example 3-tuple follows:
  1. The 3-tuple $< l_1, l_2.class, l_3 >$ of lock names, where l_1, $l_2.class$ and l_3 are names of nodes of the `tlog`, and (of course) l_1, l_2 and l_3 are names of types.
  2. A collection <Thread1-set> of thread stacks, each element of which is a prediction (for instance, like $< m_1 - [l_1]m_2 - ... - [l_2.class]m_i >$) that, (*under conducive assignment of values to objects*) if there were a program thread that were to invoke method m_1, then that in turn *could* invoke m_2, which would lock *some* object of type l_1 and proceed to in turn ...

could go on to finally invoke m_i which would lock *the* object of type $l_2.class$.

3. A collection <Thread2-set> of thread stacks, each element of which is a prediction (for instance, like $< [l_2.class]n_1 - ... - [l_3]n_j >$) that, (*under conducive assignment of values to objects*) if there were a program thread that were to invoke method n_1, then that would lock *the* object of type $l_2.class$ and proceed to in turn ... *could* go on to finally invoke n_j which would lock *some* object of type l_3.
4. Similarly, a collection <Thread3-set> of thread stacks, each element of which is a prediction (for instance, like $< p_1 - p_2 - [l_3]p_3 - ... - [l_1]p_k >$) that, (*under conducive assignment of values to objects*) if there were a program thread that were to invoke method p_1, then that in turn *could* invoke p_2, which in turn *could* invoke p_3, which would lock *some* object of type l_3 and proceed to in turn ... *could* go on to finally invoke p_k which would lock *some* object of type l_1.