Superimpositions and Aspect-Oriented Programming

MARCELO SIHMAN AND SHMUEL KATZ

Department of Computer Science, Technion - Israel Institute of Technology, Haifa, Israel
Email: {sihman, katz}@cs.technion.ac.il

The ideas of a classic distributed superimposition are used to design a new object-oriented version incorporating aspects. A superimposition is a collection of generic parameterized aspects and new classes (often singleton concrete classes). Superimpositions can be combined, either sequentially or in a merge, to create new ones. Superimpositions also include specifications about assumed properties of basic programs to which the superimposition can be applied, and desired properties added by the superimposition. These specifications are used to define proof obligations for the correctness of superimpositions, and to check feasibility of combining superimpositions. SuperJ, a notation and an implemented preprocessor over AspectJ, is described. SuperJ can be used to apply a superimposition to a basic system, generating concrete aspects from generic aspects and then weaving them to basic classes. Superimpositions are separately declared, specified, and verified. Among the examples used to demonstrate the approach are a termination detection algorithm, a version of the dining philosophers algorithm, and a monitoring superimposition that gathers statistics on basic objects.

1. INTRODUCTION

1.1. Motivation

A 'classic' superimposition is a program module that can augment an underlying (often distributed) program with added functionality, while cutting across usual language modularity constructs like processes or packages. Since the aspects approach to (mainly) OOP has an identical goal of providing facilities to express cross-cutting concerns, it is natural to consider combining superimpositions and aspects, as first suggested in [1]. In this paper, a new notation and preprocessor for superimpositions - called SuperJ - is proposed in terms of aspects.

Superimpositions are redefined in this OOP context to be a collection of generic aspects and new objects (and thus can be seen as a package of aspects and new classes). Superimpositions include specifications, both of the assumptions made about basic systems to which the superimposition can be applied, and the added properties true of the resultant augmented program. These specifications are used to define proof obligations for the correctness of superimpositions, and to check feasibility of combining superimpositions to obtain new ones. By isolating the description, specification, and verification of superimpositions that describe cross-cutting algorithms with a variety of needed augmentations, the expressiveness and modularity of AOP is extended.

1.2. Classic Distributed Superimposition

Superimposition is a type of program decomposition that facilitates writing an algorithm that can be interleaved with any one of a collection of basic programs, coded separately from them. Superimposition is very natural for distributed systems, where maintaining properties of the computation is especially difficult. Well-known examples of superimpositions are termination and deadlock detection, monitoring or debugging, adding scheduling restrictions, mutual exclusion, or bounding the possible values of variables that were unbounded in the basic program. These examples have in common the need to add an algorithm over many basic programs, usually not available when the superimposition is written.

Numerous suggestions ([2, 3, 4, 5, 6, 7]) have been made for a syntax that allows augmenting program units such as processes. A brief survey about several proposals of a language construct for superimpositions may be found in [8]. Superimposition language proposals have generally not been implemented or have dealt with academic or design languages, treating basic semantic questions or descriptions of algorithms, rather than use for real systems.

A distributed superimposition language construct, as defined in [2], has declarations for new processes, and schematic abstractions of processes called *roletypes*, each with formal parameters, local variables, transformation rules that describe additions or changes to code in the program to which the superimposition is to be applied, and new code sections that communicate with the sections in other roletypes and processes to implement a distributed algorithm. Each roletype corresponds to a different type of augmentation. Thus in a classic superimposition to detect deadlock, a tree structure of message communication is superimposed, with roletypes that correspond to the root, to internal nodes, and to leaves of the tree. A *binding* (*weaving*) operator combines the superimposition with a basic program (a collection of basic processes) and produces an augmented program. At binding time, each relevant basic process is combined with a roletype, whose parameters must be bound to variables or locations of the basic process. The new processes of the superimposition are also added to the augmented program. A superimposition can be separately declared, specified, and verified.

Superimpositions have been classified into three types ([2]): *spectative*, *regulative* and *invasive*. These differ by the degree to which the superimposed code is allowed to affect the values of variables in the original program, and the influence this has on the safety and liveness properties that held in the original program.

The *spectative* versions do not affect basic variables or conditions for applying actions, and only gather additional information about the system, e.g., for monitoring purposes. *Regulative* superimpositions spectate and also affect which actions can be taken at each stage, but do not change the actions themselves, as far as they relate to the basic variables. Finally, *invasive* superimpositions do change the actions themselves. The weaker types can do less, but guarantee by construction that important classes of properties (e.g, all safety properties) are maintained after the superimposition. The more powerful invasive type may require reproving that properties are not disturbed when the original system is combined with such a superimposition. These distinctions carry over to the superimpositions to be defined here, and will affect the proof obligations.

1.3. Aspect-Oriented Programming

The motivation behind AOP [9] is that existing programming languages cannot represent independent concerns like synchronization, resource sharing, distribution, or debugging in a module. Rather than being localized within a program unit, like a class, these concerns are orthogonal to the system's basic program units and module structure.

An aspect is therefore a new object-oriented program unit whose main goal is to allow a programmer to write a code segment that cuts across existing class structures and basic units in aspects, rather than as multiple code fragments tangled throughout the basic units. The implementation of an AOP-based system consists of a component language for writing the basic program, one or more aspect languages for writing the aspects, and an aspect weaver for the combined languages, to bind the basic program with the aspects. The weaver combines the aspects with the basic program and generates the augmented program. The binding of the aspects with the basic program is specified in the aspects by join points, which determine where the aspect should augment the basic program.

A variety of ideas and notations are connected to aspects [9, 10, 11, 12, 13, 14]. The composition filters object model [14] provides control over messages (method calls) received and sent by an object. The join points are the dynamic message sends and receives arriving at an object, and most of the weaving happens at runtime. Hyper/J [13] is an aspect-oriented extension to Java, where a *hyperslice* contains all the classes and methods related to a concern. Hyperslices may be composed, then forming a *hypermodule*, which is the resulting program. Hyper/J's remodularization and composition functionality helps in handling software evolution issues. There are other approaches, not connected to AOP, which deal with the software evolution problem by means of dynamic reconfiguration of components, as in [15].

In [16], an AOP logic meta-programming framework (based on Prolog) is presented. Join points are defined by predicates on abstract syntax trees, and composition rules are defined for specific aspects. In [17], an event-based AOP approach is proposed, where join points are defined by *event patterns*, which when detected lead to associated actions.

AspectJ [11] is also an extension to Java, where the form of an aspect is similar to the form of a class, and an aspect can be applied to several classes. The fact that it is written in Java - a widespread language - and that we also program aspects in Java, only having to write some additional declarations, is a major advantage. For these reasons, we have chosen AspectJ and its weaver as the aspect language/weaver used to help implement our proposed construct for superimpositions. In the continuation, we assume familiarity with the terminology and notation of Java, and only briefly introduce notation from AspectJ.

Note that we use AspectJ version 1.0.3 in this paper (since the syntax and semantics may differ between versions). In AspectJ new code to be added to a basic program can be described either using *introduction* (for new variables, methods, or constructors of an underlying class) or *advice* (for new code before or after join points). When using *introduction*, one may add new variables, methods and constructors to an underlying class. When using *advice*, one may insert new code before or after join points. Typical join points are constructor and method calls, exception throwing, variable accesses, and updates. The join points are specified by *pointcuts* that can be defined in the aspects using keywords like *call* or *within*. Aspects may also contain local methods, variables and classes, as well as constructors, since we may generate aspect instances. In the Conclusions, SuperJ is compared to AspectJ and some of the other existing notations.

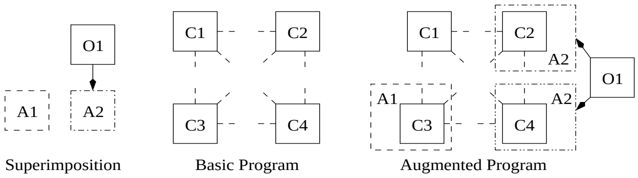

FIGURE 1. A schematic superimposition application.

2. SUPERJ

2.1. Overview

SuperJ is the name of the new construct and preprocessor for superimpositions in terms of aspects. We define a superimposition as a collection of generic parameterized aspects and new classes (often singleton concrete classes). A superimposition implements an algorithm that may be applied to different basic systems, which are composed of basic classes. Generic aspects are semantically similar to roletypes: each relevant class instantiated in the basic underlying system is bound (woven) to a generic aspect. The singleton concrete classes define unique objects that are instantiated in an augmented system, and new classes can be instantiated and used by the aspects. These objects interact with the generic aspects. As will be shown, superimpositions can be separately declared, specified, and verified, since they are not connected to any basic system.

In the version of SuperJ given here, the basic programs do not make any reference to the superimposition code, and may be executed as they are. Superimpositions add new features to a basic program, e.g. have it satisfy additional desired results. To apply a superimposition over a basic program, we bind all the superimposition parameters to basic program units like classes, methods or variable names. Thus, after the binding, the code of the augmented program that comes from the superimposition makes references to the state components of the basic program, but the opposite does not happen. A schematic representation of applying a superimposition with two aspects and a singleton class to a basic program is shown in Figure 1.

A superimposition declaration contains the declarations of its generic parameterized aspects and new classes. In addition, it contains the specification of its applicability conditions and desired result assertions. Similarly, each generic aspect contains assumptions about the classes to which it may be bound, and new properties of the augmented class. The specification of the superimposition and its generic aspects, shown in more detail in Section 2.3, can be used to prove the correctness of the superimpositions and to check the legality of their combinations.

2.2. Generic Aspects

Generic aspects have local declarations, and a series of advice and introduction sections that shall be applied to classes of basic programs. A generic aspect has no built-in connection with any program unit of any basic underlying system, and parameters are used to provide such a connection when a basic class is bound to the aspect. Global parameters appear in a list of the superimposition itself, in addition to the parameter list of each aspect.

Existing AOP constructs do not emphasize this separation. In AspectJ, for example, we need to use the names of real basic program units like class, method or variable names directly in the aspect. Thus, when identifying an instance in an advice of AspectJ, we write the name of its basic class directly in the advice, with no parameterization. This kind of restriction prevents us from defining generic aspects directly in AspectJ.

Besides its header and specification, the constructs of AspectJ, and added parameter lists, SuperJ has a keyword BOUND_CLASS which is like *this* of Java, indicating the class to which this instance of the aspect is bound. Formal parameters are in capital letters, to distinguish them from local variables.

2.3. Specifications

For each superimposition, we specify applicability conditions and desired result assertions, which serve to restrict its applicability only to basic programs that satisfy the assumptions, and to specify the new properties that hold for the augmented program, respectively. A superimposition specifies its applicability conditions in its **global condition** section, and the *desired results* added to the basic program in its **global result** section. Similarly, each generic aspect contains two sections called **condition** and **result**. The global conditions restrict the collection of basic programs and global assumptions such as the degree of synchrony in the message passing primitives used in the basic program. The local condition of a generic aspect restricts the classes that can be bound to the aspect itself or the other elements of the basic class that can be bound to the aspect's parameters.

The global results describe the added properties that the superimposition provides to the augmented program. The result section of an individual aspect can describe the added properties of the classes to which it is bound in the augmented program. There is an implicit assumption that the superimposition does not invalidate the desired properties from a specification of the basic program, so that the augmented program should satisfy both the original specification of the basic program, and the desired results from the superimposition. This is further explained in Section 4.

In the examples, the superimposition's assumptions and results are presented in words (making use of some defined predicates) and logical expressions (making use of logical definitions and predicates). Often, only part of the specification is given. However, in the proof conditions and feasibility checks that determine correctness, it is assumed that the specifications can be expressed in temporal logic ([18]), the normal specification language for verification tools, expressing properties of the possible execution sequences of the program. Therefore, an expression such as 'eventually', may be replaced by the corresponding temporal logic operator in a more formal version appropriate for verification.

2.4. Preprocessor

We have implemented a preprocessor for translating SuperJ to AspectJ (and then Java) that applies a superimposition to a basic system, generating concrete aspects for generic aspects by replacing formal parameters by basic variables, locations, and classes, and then weaving them to the relevant basic classes using AspectJ's weaver. The preprocessor, called **sj2aj**, receives two parameters from the command line: the name of the superimposition that we want to apply, and the name of a binding file whose lines each contain a binding between a basic class and a generic aspect.

Each line of *bindfile* contains the name of the basic class to be superimposed, the name of the generic aspect chosen and the actual parameter list, which gives the names of the program units that are to be associated with the formal parameters of the generic aspect chosen. It is not possible to bind several basic classes to the same generic aspect in only one line, since their actual parameter lists are usually different. Moreover, AspectJ has some characteristics that invalidate this alternative, e.g., the need to declare the real name of a basic class directly in the advice when identifying an instance of it. Thus, in a generic aspect we have to define the number of parameters used in the advice.

The preprocessor generates a concrete aspect for each generic aspect bound to a basic class in *bindfile*. For each binding in *bindfile*, **sj2aj** creates the concrete aspect generated by replacing the formal parameters by the actual parameters. Often the same generic aspect is bound to many basic classes. After building all the generated concrete aspects, **sj2aj** uses AspectJ's weaver to compile all the concrete aspects and basic classes, and produces the augmented system.

The use of AspectJ in SuperJ's implementation introduces some undesirable restrictions to SuperJ, e.g., the impossibility to bind several basic classes to the same generic aspect all at once. Since these drawbacks are not intrinsic to SuperJ's design, we may overcome them by replacing AspectJ by a dedicated implementation. In this new implementation, we would be able to apply a generic aspect over several basic classes all at once, as well as removing other technical restrictions. Alternatively, we could also base SuperJ's implementation on ConcernJ, which is presently being developed as the AOP language for composition filters. This would still give the advantages of grouping aspects and using specifications. In addition, it is also possible to use SuperJ in the design stage of an aspect-oriented system, rather than in a programming language implementation.

superimposition TerminationDetection {
global result (Source.finished $\Rightarrow$ basic program is done) and
  (basic program is done $\Rightarrow$ eventually Source.finished)
global condition nodes in CHILDREN form a spanning tree,
  $Q \in CHILDREN_P \Rightarrow P = PARENT_Q$,
  semi-synchronous communication

aspect Source(MSG_METHODS,IDLE,CHILDREN)
  extends Node issingleton();
condition bound to root of spanning tree,
  IDLE $\Rightarrow$ new basic computation only if a message is received

aspect Internal(MSG_METHODS,IDLE,PARENT,CHILDREN)
  extends Node perthis(within(BOUND_CLASS));
condition bound to internal node of spanning tree,
  IDLE $\Rightarrow$ new basic computation only if a message is received

aspect Leaf(MSG_METHODS,IDLE,PARENT)
  extends Node perthis(within(BOUND_CLASS));
condition bound to leaf of spanning tree,
  IDLE $\Rightarrow$ new basic computation only if a message is received
}

FIGURE 2. The superimposition declaration of TerminationDetection.

3. EXAMPLES OF SUPERIMPOSITIONS

3.1. Termination Detection

The superimposition TerminationDetection, whose declaration is shown in Figure 2, implements a version of a termination detection algorithm due to Topor ([19]). In the original version, new actions are interspersed with the regular actions in a system to ensure proper termination of a distributed system that otherwise could deadlock with each relevant process waiting indefinitely to serve the others. The idea of the superimposition is that a 'wave' of colored tokens will travel from the leaves to the root of a spanning tree of the objects. If the basic program has not terminated, this will be sensed in the root, and a *repeat* wave is sent to the leaves, so that the token wave can be reinitialized. The superimposition here contains three generic aspects: Source must be bound to the root of the spanning tree, Internal to the internal nodes, and Leaf to the leaves. These generic aspects and the abstract aspect Node, which they inherit, are shown in Figure 3.

The information on whether termination has occurred is encoded into the color of the tokens and a *node* variable from each generic aspect. Whenever a node sends a (basic) message, it will color its *node* variable black, in order to denote that it may have disturbed one of the nodes that was previously idle. In the superimposition, we find this node coloring in the only advice of the abstract aspect Node, which is extended by the three generic aspects. Node's single advice is:

```
pointcut send(): sendClass() && sendMethods();
after(): send() {
  node = black;
}
```

A single abstract aspect like Node is used in SuperJ

```
abstract aspect Node {
  public static final int black = 0;
  public static final int white = 1;
  protected int token = white, node = white;

  abstract protected pointcut sendClass();
  abstract protected pointcut sendMethods();

  pointcut send(): sendClass() && sendMethods();
  after(): send() {
    node = black;
  }
}

aspect Source(MSG_METHODS,IDLE,CHILDREN)
  extends Node issingleton() {

  protected pointcut sendClass(): within(BOUND_CLASS);
  protected pointcut sendMethods(): call(MSG_METHODS);

  private boolean finished = false;
  private int numbTokens = 0;

  declare parents: BOUND_CLASS implements SourceIF;
  public void BOUND_CLASS.tokenmsg(int x)
    throws RemoteException {}
  //Source's single advice
  after(BOUND_CLASS C, int x): target(C) && args(x) &&
        execution(void SourceIF.tokenmsg(int)) {
    if (x == black)
      token = black;
    numbTokens++;
    if (C.CHILDREN.length == numbTokens) {
      if (!C.IDLE || (node == black) || (token == black)) {
        token = white;
        node = white;
        numbTokens = 0;
        try {
          for (int i=0; i < C.CHILDREN.length; i++) {
            try {
              ((InternalIF) C.CHILDREN[i]).repeat();
            } catch (ClassCastException e) {
              ((LeafIF) C.CHILDREN[i]).repeat();
            }
          }
        } catch (Exception e) {
          System.err.println("Source err: " + e.getMessage());
          e.printStackTrace();
        }
      } else {
        finished = true;
      }
    }
  }
}

aspect Internal(MSG_METHODS,IDLE,PARENT,CHILDREN)
  extends Node perthis(within(BOUND_CLASS)) {

  protected pointcut sendClass(): within(BOUND_CLASS);
  protected pointcut sendMethods(): call(MSG_METHODS);

  private int numbTokens = 0;

  //Internal's first advice
  after(BOUND_CLASS C): set(boolean BOUND_CLASS.IDLE)
                        && target(C) {
    checkRule(C);
  }

  declare parents: BOUND_CLASS implements InternalIF;
  //Internal's second advice
  public void BOUND_CLASS.tokenmsg(int x)
    throws RemoteException {}
  after(BOUND_CLASS C, int x): this(C) && args(x) &&
        execution(void InternalIF.tokenmsg(int)) {
    if (x == black)
      token = black;
    numbTokens++;

    checkRule(C);
  }

  private void checkRule(BOUND_CLASS C) {
    try {
      if (C.IDLE && (C.CHILDREN.length == numbTokens)) {

        numbTokens = 0;
        if (token == white) {
          try {
            ((InternalIF) C.PARENT).tokenmsg(node);
          } catch (ClassCastException e) {
            ((SourceIF) C.PARENT).tokenmsg(node);
          }
        } else {
          try {
            ((InternalIF) C.PARENT).tokenmsg(black);
          } catch (ClassCastException e) {
            ((SourceIF) C.PARENT).tokenmsg(black);
          }
        }
        token = white;
        node = white;
      }
    } catch (Exception e) {
      System.err.println("Internal err: " + e.getMessage());
      e.printStackTrace();
    }
  }

  public void BOUND_CLASS.repeat() throws RemoteException {}
  //Internal's third advice
  after(BOUND_CLASS C): execution(void InternalIF.repeat())
                        && target(C) {
    try {
      for (int i=0; i < C.CHILDREN.length; i++) {
        try {
          ((InternalIF) C.CHILDREN[i]).repeat();
        } catch (ClassCastException e) {
          ((LeafIF) C.CHILDREN[i]).repeat();
        }
      }
    } catch (Exception e) {
      System.err.println("Internal err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}

aspect Leaf(MSG_METHODS,IDLE,PARENT)
  extends Node perthis(within(BOUND_CLASS)) {
  protected pointcut sendClass(): within(BOUND_CLASS);
  protected pointcut sendMethods(): call(MSG_METHODS);
  //Leaf's first advice
  after(BOUND_CLASS C): set(boolean BOUND_CLASS.IDLE) &&
        this(C) && !cflow(initialization(BOUND_CLASS.new(..))) {
    checkRule(C);
  }

  declare parents: BOUND_CLASS implements LeafIF;
  //Leaf's second advice
  public void BOUND_CLASS.repeat() throws RemoteException {}
  after(BOUND_CLASS C): execution(void LeafIF.repeat())
                        && this(C) {
    token = white;
    checkRule(C);
  }

  private void checkRule(BOUND_CLASS C) {
    try {
      if (C.IDLE && (token == white)) {
        try {
          ((InternalIF) C.PARENT).tokenmsg(node);
        } catch (ClassCastException e) {
          ((SourceIF) C.PARENT).tokenmsg(node);
        }
        token = black;
        node = white;
      }
    } catch (Exception e) {
      System.err.println("Leaf err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

FIGURE 3. A superimposition for termination detection.

to contain definitions, such as variable and advice declarations, that are common to all the generic aspects. Node also contains the definitions of the constant variables *black* and *white*, and the declarations of the variables *token* and *node*, which are common to all the generic aspects.

In Node, we define two abstract pointcuts (of AspectJ): *sendClass*() and *sendMethods*(). These pointcuts are associated, respectively, to the bound basic class and the remote methods used for sending messages, which are represented in the generic aspects by SuperJ's reserved word BOUND_CLASS (representing the basic class bound to the generic aspect) and by the MSG_METHODS parameter. Each time a remote message method is called from an instance of the bound class, the *node* variable of the aspect is colored black. Thus, the (implicit) requirement that the *node* variable must be set to *black* whenever a message is sent (and before IDLE becomes true) is guaranteed.

In the beginning of the algorithm execution, an idle leaf starts sending the color of its *node* variable to its parent, which is stated in Leaf's first advice by calling to its *checkRule* method right after the actual parameter bound to IDLE is assigned a value in the constructor of the bound class. Leaf's *checkRule* method may be seen in Figure 3. Leaf's first advice is shown below:

```
after(BOUND_CLASS C):
 set(boolean BOUND_CLASS.IDLE) && this(C) &&
 !cflow(initialization(BOUND_CLASS.new(..))) {
  checkRule(C);
}
```

Leaf will send *tokenmsg* to its parent only when the actual parameter bound to IDLE is set to true. An internal node will pass on a white token only if it has received a white token from all its sons and is itself white, and otherwise will send a black token after receiving tokens from all its sons. This can be seen in the first advice and the *checkRule* method of Internal (see Figure 3). Internal's first advice is:

```
after(BOUND_CLASS C): target(C) &&
   set(boolean BOUND_CLASS.IDLE) {
  checkRule(C);
}
```

If the root is white and has received white tokens from all its sons, then termination has been detected. This check takes place in Source's single advice, after receiving a *tokenmsg*, as may be seen in Figure 3.

The connection of the basic computation to the superimposition is through the formal parameter IDLE, while the needed spanning tree is encoded into the parameters PARENT and CHILDREN. We assume - in all generic aspects' **condition** sections - that if the variable that will be bound to IDLE is true, then the (augmented) basic algorithm in the associated object has no pending internal computation and will only respond to new incoming messages. Thus this termination detection algorithm is especially appropriate for underlying reactive object systems. As may be seen in the **global result** section, the variable *finished* in the *Source* generic aspect indicates when a global passive state has been detected. This superimposition is spectative, because it merely *detects* termination of basic activities by setting a new variable, but does not change the basic computation. It could be used in conjunction with other superimpositions that announce the problem, or actually terminate the basic objects.

We use RMI (remote method invocation) to implement message passing between the objects in our examples. We define interfaces SourceIF, InternalIF and LeafIF for the generic aspects Source, Internal and Leaf, respectively; both SourceIF and InternalIF contain method *tokenmsg*, which treats the receipt of token messages, while InternalIF and LeafIF contain method *repeat*, to handle the receipt of *repeat* messages from the root. Each generic aspect introduces - to its bound class - a declaration that the augmented class implements the associated interface, and the implementation of the associated remote methods *tokenmsg* and *repeat*. This technique may be used whenever we want to introduce new messages to basic classes, often allowing us to increase their parallelism and distribution.

A possible activation of **sj2aj** for superimposition TerminationDetection for a given binding file called *bindfile* is:

sj2aj TerminationDetection *bindfile*

where a simple example of *bindfile* (with the basic class and its actual parameters in regular font, and the aspect name in slant), is:

Basic1 *Source*(*void* msg1(),inert,children,ended)
Basic2 *Internal*(*int* msg2(),unoccupied,up,down)
Basic3 *Leaf*(*void* msg1() || *void* msg3(*int*),inactive,father)

Note that in the **global condition** section the phrase 'semi-synchronous communication' appears. This is intended to mean that the basic program, as well as the augmented one, must use either synchronous *send* and *receive* operations, or at least guarantee that a message will be sensed by the receiving object before an entire wave can traverse the spanning tree. If it did not hold, the algorithm could be shown to be incorrect for basic messages that remain in nontree channels for an entire round of a token wave and a repeat wave. Since we use RMI for message passing, we are assured that this condition holds, as an object suspends its execution when calling a remote method, which is equivalent to a synchronous communication. The correctness of the superimposition is proven in [19], although the crucial assumption on semi-synchronous communication is expressed somewhat vaguely.

3.2. Dining Philosopher Problem

Figure 4 shows a superimposition (DPP-H) that solves the heavy-load case of the Dining Philosophers Problem (DPP) [20] for arbitrary neighbor graphs, where each philosopher immediately wants to eat again after finishing an eating session. This can be visualized using a constraint graph, where each augmented object is represented by a node. Two nodes P and Q are connected if and only if $P \in \text{NEIGHBORS}_Q$ and $Q \in \text{NEIGHBORS}_P$, where NEIGHBORS is an array containing a node's neighbor objects, which is a parameter of DPP-H's only generic aspect (DPP-Heavy). In DPP, a philosopher (object) may eat (execute some critical computation CC) only when it has all its associated forks. Each fork is associated with two philosophers, and must be used by only one philosopher at a time. DPP-H ensures - in the **global result** section - that augmented objects that are neighbors in the constraint graph may not execute their CC's concurrently, as each augmented object will alternate with its neighbors the right to execute its CC. Besides mutual exclusion of CC's among neighbor processes, DPP-H ensures absence of deadlock and starvation (concerning the execution of the CC's). DPP-H requires - in the **global condition** section - the NEIGHBORS arrays of the basic objects to be consistent, where the (bound) NEIGHBORS arrays are consistent when $P \in \text{NEIGHBORS}_Q \iff Q \in \text{NEIGHBORS}_P$, for any bound objects P and Q. DPP-H is very useful when a system must be upgraded to support resource sharing or new synchronization/scheduling restrictions among its component objects.

The aspect adds code to initially orient the graph acyclically, by exploiting the ids of the basic objects (using the MY_ID and NEIG_IDS parameters), which is implemented by DPP-Heavy's first advice, and added to the end of the basic class' constructor. It also adds the check that the node has all its associated forks in DPP-Heavy's second advice when calling the *checkGuard* method before executing EAT_METHOD, where EAT_METHOD is a parameter that must be bound to the basic class' critical method.

The aspect also deals with the sending and receiving of the forks, in the second and third advice. In the third advice, when an augmented object P receives a fork (via a remote call to its introduced method *forkmsg()*), it increments its *forks* counter. In the second advice, after finishing to execute its CC, P sends all its forks to its neighbors and resets *forks* to zero. DPP-Heavy requires - in the **condition** section - the basic object to run continuously and to try to start executing its CC consecutively. DPP-Heavy ensures - in the **result** section - absence of deadlock and starvation regarding the execution of P's CC (since P may execute its CC infinitely often).

DPP-H changes the execution sequences of the basic program, but it does not modify its state space, so it is classified as a *regulative* superimposition. Even though there is only one aspect, the genericity is crucial so that it can be reused for different systems, or even in different connected components of one system.

```
superimposition DPP-H {
global result mutual exclusion; absence of deadlock and starvation
  P and Q may execute CC concurrently ⟺ ¬neighbors(P,Q)
  each object P may execute its CC infinitely often
global condition P ∈ NEIGHBORS_Q ⟺ Q ∈ NEIGHBORS_P

aspect DPP-Heavy(EAT_METHOD,AN_INTERFACE,NEIGHBORS,
NEIG_IDS,MY_ID) perthis(within(BOUND_CLASS));
result aug obj P will obtain all its forks in finite time infinitely often
condition P continues running; P continuously calls EAT_METHOD
}

aspect DPP-Heavy(EAT_METHOD,AN_INTERFACE,NEIGHBORS,
NEIG_IDS,MY_ID) perthis(within(BOUND_CLASS)) {
  private int forks = 0;

  private void checkGuard(BOUND_CLASS C) {
    synchronized(C) {
      while (forks != C.NEIGHBORS.length)
        try {C.wait();}
        catch (InterruptedException e) {}
    }
  }

  //first advice
  after(BOUND_CLASS C): initialization(BOUND_CLASS.new(..))
                        && target(C) {
    for (int i=0; i < C.NEIG_IDS.length; i++)
      if (C.MY_ID.compareTo(C.NEIG_IDS[i]) > 0)
        forks++;
  }

  //second advice
  void around(BOUND_CLASS C): execution(EAT_METHOD)
                              && target(C) {
    checkGuard(C);
    proceed(C);
    for (int i=0; i < C.NEIGHBORS.length; i++) {
      try {
        ((HeavyIF) C.NEIGHBORS[i]).forkmsg();
      } catch (Exception e) {
        System.err.println("Heavy err: " + e.getMessage());
        e.printStackTrace();
      }
    }
    forks = 0;
  }

  declare parents: BOUND_CLASS implements HeavyIF;
  public void BOUND_CLASS.forkmsg() throws RemoteException {}

  //third advice
  void around(BOUND_CLASS C): execution(void HeavyIF.forkmsg())
                              && target(C) {
    forks++;
    synchronized(C) {
      C.notify();
    }
  }
}
```

FIGURE 4. A superimposition for the heavy-load case of the DPP.

3.3. Monitoring Execution

Figure 5 shows a trivial superimposition (Monitoring) that gathers statistics on basic objects, such as counting the total number of external method calls for all relevant basic objects. An external method call to an object takes place when one of its methods is called, and this call is not from inside one of the object's methods. The Monitoring superimposition does not modify either the values of the variables, or the possible execution sequences of the basic program, so it falls in the *spectative* category of superimpositions. The superimposition contains two generic aspects (Constant and Mutable) and one singleton class (Coordinator). Constant and Muta-

```
superimposition Monitoring {
global result coord.totCalls = sum of all external method calls of
   terminated objects, coord.totConCalls = sum of external method
   calls of terminated constant objects ···

aspect Constant(END_METHOD) extends Common
perthis(within(BOUND_CLASS));
result nCalls = sum of external method calls to the object
   nAssigns = sum of assignments performed in the object
condition END_METHOD = last method executed in the object

aspect Mutable(END_METHOD) extends Common
perthis(within(BOUND_CLASS));
result nCalls = sum of external method calls to the object;
   nAssigns = sum of assignments performed in the object
condition END_METHOD = last method executed in the object

class Coordinator coord;
}

class Coordinator {
   private int totCalls = 0;
   private int totConCalls = 0;
   private int totMutCalls = 0;
   private int totConAssigns = 0;
   private int totMutAssigns = 0;
   private DataOutputStream conLogFile;

   public void conMethodCount(int x) {
     totConCalls += x; totCalls += x;
   }
   public void mutMethodCount(int x) {
     totMutCalls += x; totCalls += x;
   }
   public void conAssignError(int x, Vector log) {···}
   public void mutAssignCount(int x) {totMutAssigns += x;}
}

abstract aspect Common {
   protected final static Coordinator coord = new Coordinator();
   protected int nCalls = 0;
   protected int nAssigns = 0;

   abstract protected pointcut allExternalCalls();
   after(): allExternalCalls() {
     nCalls++;
   }
}

aspect Constant(END_METHOD) extends Common
perthis(within(BOUND_CLASS)) {
  private Vector log;
  protected pointcut allExternalCalls(): !cflowbelow
  (within(BOUND_CLASS)) && execution(* BOUND_CLASS.* (..));
  after(): set(* BOUND_CLASS.*) &&
           !cflow(initialization(BOUND_CLASS.new(..))) {
    nAssigns++;
    logRecord();
  }
  after(): execution(* BOUND_CLASS.END_METHOD(..)) {
    coord.conMethodCount(nCalls);
    coord.conAssignError(nAssigns, log);
  }
  private void logRecord() {···}
}

aspect Mutable(END_METHOD) extends Common
perthis(within(BOUND_CLASS)) {
  protected pointcut allExternalCalls(): !cflowbelow
  (within(BOUND_CLASS)) && execution(* BOUND_CLASS.* (..));
  after(): set(* BOUND_CLASS.*) {
    nAssigns++;
  }
  after(): execution(* BOUND_CLASS.END_METHOD(..)) {
    coord.mutMethodCount(nCalls);
    coord.mutAssignCount(nAssigns);
  }
}
```

FIGURE 5. A monitoring superimposition.

ble extend the Common abstract aspect, which contains code common to both generic aspects. The Common aspect uses the Coordinator class and creates its single instance *coord*, which is used by Constant and Mutable; moreover, its advice increments the (local) *nCalls* counter after each external call to any method of the bound class, (so each augmented object will have its own *nCalls* counter). After each field assignment performed in the bound (basic) object, Mutable's advice increments the local *nAssigns* counter. The only instance of Coordinator (*coord*) accumulates the global statistics gathered by Constant and Mutable. The Constant aspect is applied to (constant) classes whose field values should not be changed. When a field assignment is nevertheless tried, the aspect increments the *nAssigns* counter, but also records locally that a violation of the desired feature of the field being constant has occurred.

Each basic object augmented by Mutable will call *coord*'s *mutMethodCount* and *mutAssignCount* methods, while objects augmented by Constant will call *conMethodCount* and *conAssignError*. These methods increment a global counter, and specific counters for each type, while the ones from Constant also transfer the recorded information and produce a log of the assignments to objects supposedly constant. Of course, Monitoring could make more sophisticated use of the gathered statistics. Generalizations of the same idea should be useful for bookkeeping and debugging. In particular, superimposition is especially appropriate when the generic aspects have more interaction, as when the statistics collected by each generic aspect are combined.

4. PROVING CORRECTNESS

In this section, we establish the foundations for proving correctness of superimpositions in AOP. We consider both the correctness of the result of applying a superimposition to a given basic program with a specific binding of parameters, and the more abstract correctness of the superimposition itself. For this purpose, we use some definitions and notations introduced in [2], and show a collection of proof obligations. The proof obligations are expressed as 'regular' programs and specifications to be verified using known software verification methods, such as model checking or inductive theorem proving. For example, Java model checking can be done using tools such as Bandera [21]. That tool verifies general temporal logic properties by defining finite state abstractions of the programs, and automatically translating to the input notation of one of several existing model checking tools, which are then used for the verification. Infinite state programs, with unbounded data, can be treated using inductive theorem proving methods, for example in the STeP [22] system or PVS [23], that both have associated C-like interface languages. Although verifying Java programs is still a research topic, here we view this as a given technology, since proof methods for general software is beyond the scope of this paper.

DEFINITION 1. An *augmented program* (i.e., the result of applying a superimposition to a basic program) *is correct* if it satisfies both the desired specification of the basic program and the result assertions of the superimposition.

Note that we do not want the augmented program to preserve *all* the properties that the basic one satisfied before being bound to the superimposition, as it may have a property, like deadlock, that now we want to remove. So, the properties that must be preserved are only the desirable ones, given in the specification of the basic program. These should not be affected by the superimposition.

Definition 2. A *superimposition is correct* if, for any basic program that satisfies all the assumptions and requirements made by the superimposition globally and locally, the augmented program obtained by any legal binding of the superimposition to that basic program is correct.

To express this in a concise notation, a superimposition is represented as a triple $(\mathbf{S},\mathbf{C},\mathbf{R})$, where $\mathbf{S}$ is the superimposition code, $\mathbf{C}$ its applicability conditions and restrictions on generic aspect bindings, and $\mathbf{R}$ its desired result assertions. A basic program is a pair $(\mathbf{B},\mathbf{P})$, where $\mathbf{B}$ is the code of the program, and $\mathbf{P}$ a specification of its desirable properties. The notation $\mathbf{x}\leftarrow\mathbf{y}$ is used to denote the replacement of formal parameters $\mathbf{x}$ of the superimposition by elements $\mathbf{y}$ (variables, objects, methods, locations, etc.) of $\mathbf{B}$. Then a superimposition $(\mathbf{S},\mathbf{C},\mathbf{R})$ is correct, if whenever applied to $\mathbf{B}$ with specification $\mathbf{P}$, where $\mathbf{B} \Rightarrow \mathbf{P}$ and $\mathbf{B} \Rightarrow \mathbf{C}[\mathbf{x}\leftarrow\mathbf{y}]$, then

$$(\mathbf{B} + \mathbf{S}[\mathbf{x}\leftarrow\mathbf{y}]) \Rightarrow \mathbf{R}[\mathbf{x}\leftarrow\mathbf{y}] \wedge \mathbf{P}.$$

The meaning of the expression $(\mathbf{B} + \mathbf{S}[\mathbf{x}\leftarrow\mathbf{y}])$ is the augmentation of $\mathbf{B}$ with superimposition $\mathbf{S}$, where the parameters $\mathbf{x}$ of $\mathbf{S}$ are replaced by the elements $\mathbf{y}$ of $\mathbf{B}$. Moreover, a semantic implication like $\mathbf{X} \Rightarrow \mathbf{Y}$ means that the program or assertion $\mathbf{X}$ defines sequences of computations that are a subset of the ones defined by $\mathbf{Y}$. Hence, if the basic program satisfies its own specification and the applicability conditions of the superimposition, then the augmented program derived from the application of $\mathbf{S}$ over $\mathbf{B}$ will satisfy the result assertions, with $\mathbf{x}$ replaced by $\mathbf{y}$, and still maintain the original desired properties $\mathbf{P}$.

In expressing the ideas above as instances of proof obligations, it is often desirable to divide the proofs into parts, so that tracing the source of a mistake becomes easier. For a specific augmented program, we should show that:

1. The basic program $\mathbf{B}$ satisfies its specification $\mathbf{P}$
2. The basic program $\mathbf{B}$ satisfies the applicability conditions of the superimposition $\mathbf{C}[\mathbf{x}\leftarrow\mathbf{y}]$
3. The augmented program $(\mathbf{B} + \mathbf{S}[\mathbf{x}\leftarrow\mathbf{y}])$ satisfies the basic specification $\mathbf{P}$
4. The augmented program satisfies the result assertions of the superimposition $\mathbf{R}[\mathbf{x}\leftarrow\mathbf{y}]$

Note that if the first two items have been proven, they can be used as assumptions in proving the last two, especially if theorem proving techniques are used. Otherwise, showing (2) is extraneous, although it is unlikely that (3) and (4) would hold if (2) is false.

However, it is obviously impossible to verify the correctness of a superimposition by considering every possible augmented program explicitly. Instead we would like to do one 'generic' proof, even though a superimposition by itself is not a program. Such a proof does consider an arbitrary, unknown, basic program, $\mathbf{D}$ which is assumed to satisfy the applicability conditions defined by the superimposition in $\mathbf{C}$. Then we must prove that if both $\mathbf{C}$ and $(\mathbf{D} \Rightarrow \mathbf{C})$ are assumed, then

$$(\mathbf{D} + \mathbf{S} \Rightarrow \mathbf{R})$$

holds. We do not know anything about the dummy program $\mathbf{D}$, except that it satisfies the applicability conditions $\mathbf{C}$ of the superimposition. We need to prove that when combined with the superimposition it should satisfy the result assertions $\mathbf{R}$.

Such a proof can be done in an inductive framework, where $\mathbf{D}$ represents any more specific program that satisfies the assumptions. Thus if the proof succeeds, the superimposition must be correct. A model checking proof can also be done, if the proper abstraction of all basic programs can be found. The abstract program must have classes and states that satisfy the assumptions of the superimposition, and also states that correspond to predicates tested by the superimposition or locations that can be reached. That is, if a predicate is tested whenever a (parametric) method is called, the abstract program should have a state where the predicate is false when a (corresponding concrete) method is called, and another where it is true. This is analogous to the abstraction seen in usual Bandera verifications, where only the 'significant' differences are maintained. It is also related to work on model checking a representative model built from a model-generating graph grammar and then concluding that any model that can be generated from the grammar will be correct [24].

As an example, the TerminationDetection superimposition assumes that each basic object bound to one of the generic aspects must have a variable that corresponds to IDLE in that it is set to true only when the program has no active computation and becomes passive. When the condition is true, correctness can be proven, while otherwise, the termination detection is clearly not guaranteed.

Using the symbols defined previously, we have

Claim 1. If superimposition $\mathbf{S}$ is correct and is applied to a correct basic program $(\mathbf{B},\mathbf{P})$ with binding $\mathbf{x}\leftarrow\mathbf{y}$ where $\mathbf{B} \Rightarrow \mathbf{C}[\mathbf{x}\leftarrow\mathbf{y}]$ and $(\mathbf{B} + \mathbf{S}[\mathbf{x}\leftarrow\mathbf{y}]) \Rightarrow \mathbf{P}$, then the augmented program is correct.

The proof of this claim follows directly from the previous definitions. Another statement of the claim is that for a correct superimposition, if we can show proof obligations (1)–(3) for a basic program and its augmented version, we can immediately also conclude (4) without further proof.

When a correct basic program appropriate for the superimposition in a legal binding is used, in order to guarantee the correctness of the combination, it remains only to show proof obligation (3), that the augmented program still satisfies the desirable properties of the basic program. If the superimposition does not change variables that are formal parameters or close possible execution paths, i.e., is *spectative*, then the states of the basic program are unchanged in the augmented version, and thus maintain the same properties, showing (3) for basic programs that indeed satisfy their own specification. All the new properties relate to the part of the state added by the superimposition. Thus for the TerminationDetection and Monitoring superimpositions, which are both *spectative*, all safety and liveness properties of any basic program to which they are applied will be maintained.

Otherwise, if the superimposition closes any execution path of the basic program (i.e., is *regulative*) or changes variables of it (i.e., is *invasive*), then it must be shown that the desirable results of the basic program are preserved, again using verification techniques.

A *regulative* superimposition may change the execution sequences of a basic program, but it may not modify its state space. The dining philosopher superimposition is of this type. The intention is to preserve the safety properties of the basic program, despite affecting its liveness properties. Even if we assume that a dummy program satisfies the assumptions required by the superimposition, and after we have proven that the desired properties of the superimposition, such as mutual exclusion among critical computations of objects that share a fork, we need to prove that the augmented version does not disturb the desired liveness properties of the basic computation, for example that every request to execute a critical computation is eventually granted.

The most liberal and powerful approach is the *invasive* one, as a superimposition may not only alter the execution sequences of the basic program, but also modify its state space by changing the values of parameters that are bound to elements of basic objects. The general proof obligations given previously apply to invasive superimpositions, and each proof obligation may be difficult to establish. The price of the added expressiveness is the need to prove (3), which can be nontrivial. However, this approach covers a collection of algorithms that can be seen as natural superimpositions and that may not be designed in either of the simpler categories. Invasive superimpositions allows the basic program to take some action based on the changes in the state of the basic program caused by the superimposition code. Then we would be able to correct a possible deadlock, bound timestamp values, and other natural superimpositions.

In deciding about the type of superimposition to be used, we need to consider cost versus benefit, for the superimposition algorithms that we want to apply. It is a decision of expressiveness versus automatic preservation of the basic program properties, and consequently simpler correctness proofs.

5. COMBINATIONS OF SUPERIMPOSITIONS

Thanks to the modularity, compositionality and reusability of superimpositions , we may now create new superimpositions by combining old ones. Two ways of combining superimpositions to create new superimpositions are considered here, although others are possible. In *sequential combinations* a new superimposition is obtained that is equivalent to first applying one, and then applying the second to the result. In *merging* combinations, it is as if each component superimposition is applied independently to a basic program, without mutual influences. In both cases the applicability conditions and the desired result assertions of the component superimpositions are compared and used to determine whether the combination is possible. If so, they are then combined along with the code of the components to obtain both the specification and the code of the resultant superimposition, without considering any specific basic program.

The generation of the cross-product of pairs of aspects from the components being combined can be done mechanically, or a specific version that does not generate all possible combined aspects can be specified. The component specifications may restrict possible combinations of superimpositions or of their internal substructures, when their specifications are contradictory. This happens when their results are contradictory, or the 'lower' superimposition gives a result that contradicts an assumption of the 'upper' one. For merging combinations, a new proof obligation is also needed, showing that one does not have code that interferes with the other. The feasibility conditions can be automatically generated, and then checked by a theorem prover or satisfiability solver to see if they are contradictory.

Such combinations help capture natural modularity and separation of concerns that otherwise are not expressible. In [8], these combinations were proposed for classic superimpositions.

It is natural to apply multiple superimpositions to a basic program, one after the other. Initially, a superimposition can be applied to a basic program, yielding an augmented program that may be seen as a new basic program. Then we apply a second superimposition to this new basic program and receive the final augmented program. Here we combine superimpositions in order to achieve a similar result in a different way. First, sequential combination applies the superimpositions one over the other (without any basic program), yielding a new combined superimposition. We then can apply the combined superimposition to a basic program. This is useful when we want to apply multiple superimpositions in the same order to many basic programs, or when the basic program is large, so that the binding itself is complex.

When using AspectJ's weaver to apply two distinct aspects to a class, each aspect influences the other, and so AspectJ's weaver does not directly support either of the suggested views of multiple superimpositions. In addition, a regular aspect's *perthis*(*pointcut*) declaration (used to create an aspect instance for each object that fits *pointcut*) affects the way one aspect influences the other in an unclear way in AspectJ. Of course, there is no provision for building up complex aspects out of simpler ones. In our implementation, the combination operators make use of AspectJ's $-preprocessor$ option, which causes the weaver to generate Java source files (instead of binary class files, which would prevent the further use of the superimpositions generated by the combination).

In sequential combination, we want the resulting superimposition (from the sequential combination of two superimpositions - B over A) to cover all possible combined generic aspects. That is, each generic aspect of A is augmented by each of B, each singleton class of A is augmented by **one** generic aspect of B, and the singleton classes of B are included as they are. For example, let A and B contain two generic aspects and one singleton class each, and we want to apply both to various basic programs, with the equivalent order of application being A over a basic program, and then B. Thus, the new superimposition B/A will have four generic aspects and two singleton classes, which is the combination of all generic aspects of B over the generic aspects of A, and the union of their singleton classes, where A's singleton class is associated with a generic aspect of B.

The sequential combination's binding mechanism for generating B/A and the internal aspects is similar to the usual binding between a superimposition and a basic program. Below we denote a typical aspect of A as gaA and an aspect of B as gaB to show some differences. Now we may bind parameters of gaB to parameters or locally defined variables of gaA. In the former case - the parameters are the 'same' - and only one (the parameter of the 'lower' generic aspect gaA) will appear in the combination gaB/gaA. In the latter case, the local variable of gaA 'discharges' the binding needed for the parameter of gaB - and thus the parameter is hidden in the combination. Additionally, all the parameters of gaA and gaB that remain unbound will appear in the parameter list of gaB/gaA.

Before combining the generic aspects, the binding mechanism needs to check if the sequential combination of B over A is feasible. In particular, it needs to check that the desired results specified in the **global result** section of A do not contradict the assumptions specified in the **global condition** section of B. Also, that the desired results of A and B are not contradictory. Other situations can also lead to contradictions. Moreover, the same mechanism is applied to check the assumptions and desired results of each pair of generic aspects - and pairs of generic aspect and singleton class - to be combined. It is possible that only a subset of the generic aspects generated by this combination will be present in B/A, as some generic aspect gaB/gaA may be eliminated because of contradictory conditions or desired results specified in gaB and gaA. However, if there is any contradiction in the combination of a singleton class of A with the generic aspect of B chosen to be bound with it, then we cannot create B/A at all, since B/A needs all the singleton classes of A and B.

Some of the global assumptions of B may be discharged by the results of A, and thus be hidden in the combination. The remaining assumptions in the **global condition** section of B/A are those not implied by the results of A, along with the assumptions of A. The desired results of B/A will contain the conjunction of the desired results of A and B.

We outline the application of the sequential combination by means of an example, which makes use of DPP-H and Monitoring. We superimpose Monitoring over DPP-H, since we want different basic programs to use DPP's scheduling for performing concurrent computations, and we also want to gather statistics and debugging information about the execution of DPP-H's added code and the original code of an associated basic program. The new superimposition Monitoring/DPP-H has two generic aspects (Constant/DPP-Heavy and Mutable/DPP-Heavy), which are all the combinations of component generic aspects. The new generic aspects will both inherit from the Common class, which contains code common to Constant and Mutable.

In Figure 6, partial code of Constant/DPP-Heavy is given, showing the changes that Constant applies to DPP-Heavy. After DPP-Heavy's *checkGuard* method finishes, the *nCalls* counter is incremented, and after two advice sections of DPP-Heavy change the value of its *forks* field, the *nAssigns* counter (from Constant) is incremented and the *logRecord* method is called. The rest of the code of Constant/DPP-Heavy has the remaining parts of Constant and DPP-Heavy. Note that Constant's advice sections remain in Constant/DPP-Heavy, and will later be applied to basic bound classes. In Mutable/DPP-Heavy, *nCalls*'s incrementation happens as above, and after each advice section of DPP-Heavy changes the value of its *forks* field, the *nAssigns* counter is incremented. Note that Constant/DPP-Heavy is not particularly useful, as we know in advance that it will log an error, because DPP-Heavy is not constant (it changes the value of its *forks* variable).

The merging combination, like sequential combination, gets two component superimpositions that we want to combine and returns a unique superimposition

```
aspect Constant/DPP-Heavy(EAT_METHOD,···,MY_ID,
  END_METHOD) extends Common
  perthis(within(BOUND_CLASS)) {
  ...
  private void checkGuard(BOUND_CLASS C) {
    ...
    nCalls++;
  }
  ...
  void around(BOUND_CLASS C): execution(EAT_METHOD) ··· {
    ...
    forks = 0;
    nAssigns++;
    logRecord();
  }
  ...
  void around(···): execution(void HeavyIF.forkmsg()) ··· {
    forks++;
    nAssigns++;
    logRecord();
    ...
  }
}
```

FIGURE 6. The Constant/DPP-Heavy generic aspect.

as the result. The main difference is that merging does not superimpose one component over another. Merging is very useful when we want to apply two superimpositions A and B to some basic program, where A and B would affect each other if we use sequential combination, and we do not want this to happen. By 'affect each other' we mean that if we sequentially combine B over A, then B would perform its task over A, possibly changing A's computation; and vice-versa for the opposite order. Thus, if we do not want this to happen, but we do want to apply both superimpositions to the basic program, then we may not use sequential combination. In this case, we must use merging, to get a superimposition B_A that will apply both A and B to the basic program without any interference between them.

B_A contains all the combinations of generic aspects of A (gaA) and B (gaB), as in sequential combination. However, in merging there is no direct connection between the component generic aspects gaA and gaB, since no advice or introduction declaration contained in gaA is applied to gaB, and vice-versa. As in sequential combination, B_A contains the union of the singleton classes of A and B, but unchanged, since they are not bound to any generic aspect. The binding of parameters between gaA and gaB is limited to defining which of their parameters are bound together (local elements cannot be bound to parameters). For each pair of associated parameters, the parameter list of the resulting generic aspect gaB_A will contain only one parameter.

The **global condition** section of B_A has the conjunction of the **global condition** sections of A and B, and the **global result** section of B_A contains the conjunction of the component **global result** sections. The same applies to all the generic aspects of B_A concerning their **condition** and **result** sections.

As in sequential combinations, the merging also checks if the combination of A and B is feasible. In addition, the merging checks that one superimposition does not interfere with the other concerning their correct functioning over a basic program. This is a new proof obligation, not present in the sequential combination of superimpositions previously proven correct. It can often be discharged trivially if the superimpositions treat different parts of the augmented state. Otherwise, a dummy program must be generated, as in the proof of correctness of a superimposition, to show that the merged superimposition achieves both of the result assertions of the components when applied over a basic program that satisfies both of their assumptions. Since this can be as difficult as the original proofs of correctness of the components, such interacting merges are probably not worthwhile.

As an illustration, we outline the application of the merging by means of the same two components used for the sequential combination. We merge Monitoring with DPP-H, since we want different basic programs to use DPP's scheduling for performing concurrent computations, and we want to gather statistics and debugging information only about the basic program. The new superimposition Monitoring_DPP-H will contain two generic aspects (Constant_DPP-Heavy and Mutable_DPP-Heavy), which are all the combinations of component generic aspects, where both generic aspects inherit from the Common class. The new generic aspects consist of the component aspects' code, where the components do not change each other. Here, Constant_DPP-Heavy has no built-in error announcements, since an error is logged only when the basic program assigns to a supposedly constant object.

Since Java does not support multiple inheritance, if the generic aspects of both component superimpositions extend two different aspects, the combination operations create a new parent aspect (with the same type of combination) out of the two original parents, which is then extended by the new generic aspects. Other solutions, such as applying the delegation pattern, may be tried to overcome this multiple inheritance limitation.

6. CONCLUSIONS

A superimposition, as a collection of generic parameterized aspects, places associated aspects in a common place. The tasks of the superimposition and its generic aspects are well defined, where formal specification is an integral part of their definitions. Thus the usage is easier, allowing each class to be bound to the appropriate component aspect. SuperJ brings a more powerful semantics to AOP, as we may express interactions and relations among generic aspects, (like the interactions among Source, Internal and Leaf in TerminationDetection) and combine collections of aspects. A collection of aspects thus has an independent semantics and proof obligations.

In AspectJ or most of the other existing languages, there is no similar way to deal with a collection of aspects. For example, if we want to write the termination detection superimposition, then we may either write Source, Internal and Leaf as distinct aspects, or write a single aspect containing all their code. In the former case, it is not clear that the aspects are connected and interact, and have a common specification. In the latter case, we lose in modularity, compositionality, specification, and a generic definition of proof obligations. Building combinations of such modules is too complex, because the component aspects have been mixed together. It seems that the existent AOP approaches tend to emphasize the elimination of entangled code and writing programs that are clearer and easier to understand, but ascribe less importance to these issues, which are essential to a clean reusable encapsulation of cross-cutting concerns.

REFERENCES

[1] Katz, S. and Gil, J. (1999) Aspects and superimpositions. In *ECOOP'99 AOP Workshop*, June, Lisbon, Portugal, LNCS 1743, pp. 308–309. Springer, Berlin. `http://trese.cs.utwente.nl/aop-ecoop99/papers/katz.pdf`.

[2] Katz, S. (1993) A superimposition control construct for distributed systems. *ACM TOPLAS*, **15(2)**, 337–356.

[3] Back, R. and Sere, K. (1996) Superposition refinement of reactive systems. *Formal Aspects of Computing*, **8(3)**, 324–346.

[4] Bougé, L. and Francez, N. (1988) A compositional approach to superimposition. In *POPL'88*, January, San Diego, CA, pp 240–249. ACM, New York.

[5] Chandy, K. and Misra, J. (1988) *Parallel Program Design - a Foundation.* Addison-Wesley, Reading, MA.

[6] Francez, N. and Forman, I. (1996) *Interacting Processes.* Addison-Wesley, Reading, MA.

[7] Järvinen, H.-M., Kurki-Suonio, R., Sakkinen, M. and Systä, K. (1990) Object-oriented specification of reactive systems. In *ICSE'90*, March, Nice, France, pp. 63–71. IEEE Press, New York. `http://disco.cs.tut.fi`.

[8] Sihman, M. and Katz, S. (2002) A calculus of superimpositions for distributed systems. In *AOSD 2002*, April, Enschede, Netherlands, pp. 28–40. ACM, New York.

[9] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997) Aspect-oriented programming. In *ECOOP'97*, June, Jyväskylä, Finland, LNCS 1241, pp. 220–242. Springer, Berlin.

[10] Czarnecki, K. and Eisenecker, U. W. (2000) Aspect-oriented decomposition and composition. In *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Reading, MA.

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. (2001) An overview of AspectJ. In *ECOOP 2001*, June, Budapest, Hungary, LNCS 2072, pp. 327–353. Springer, Berlin. `http://aspectj.org`.

[12] Lieberherr, K. J., Silva-Lepe and Xiao, C. (1994) Adaptive object-oriented programming using graph-based customization. *CACM*, **37(5)**, 94–101.

[13] Tarr, P., Ossher, H., Harrison, W. and S. M. S. Jr. (1999) N degrees of separation: Multi-dimensional separation of concerns. In *ICSE'99*, May, Los Angeles, CA, pp. 107–119. ACM, New York.

[14] Aksit, M., Wakita, K., Bosch, J., Bergmans, L. and Yonezawa, A. (1993) Abstracting object interactions using composition filters. In *ECOOP'93 Workshop on Object-Based Distributed Programming*, July, Kaiserslautern, Germany, LNCS 791, pp. 152–184. Springer, Berlin.

[15] Wermelinger, M., Lopes, A. and Fiadeiro, J. L. (2001) A graph based architectural (Re)configuration language. In *ESEC/FSE 2001*, September, Vienna, Austria, pp 21–32. ACM, New York.

[16] De Volder, K. and D'Hondt, T. (1999) Aspect-oriented logic meta programming. In *Reflection'99*, June, Heidelberg, Germany, LNCS 1616, pp. 250–272. Springer, Berlin.

[17] Douence, R., Motelet, O. and Sudholt, M. (2001) A formal definition of crosscuts. In *Reflection 2001*, November, Heidelberg, Germany, LNCS 2192, pp. 170–186. Springer, Berlin.

[18] Manna, Z. and Pnueli, A. (1992) *The Temporal Logic of Reactive and Concurrent Systems - Specification.* Springer, Berlin.

[19] Topor, R. W. (1984) Termination detection for distributed computations. *Information Processing Letters*, **18**, 33–36.

[20] Barbosa, V. and Gafni, E. (1989) Concurrency in heavily loaded neighborhood-constrained systems. *ACM TOPLAS*, **11(4)**, 562–584.

[21] Hatcliff, J. and Dwyer, M. (2001) Using the Bandera tool set to model-check properties of concurrent Java software. In *CONCUR 2001*, August, Aalborg, Denmark, LNCS 2154, pp. 39–58. Springer, Berlin.

[22] Bjorner, N., Manna, Z., Sipma, H. and Uribe, T. (2001) Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, **253(1)**, 27–60.

[23] Owre, S., Rushby, J., Shankar, N. and von Henke, F. (1995) Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, **21(2)**, 107–125.

[24] Shtadler, Z. and Grumberg, O. (1989) Network grammars, communication behaviors and automatic verification. In *AVMFSS Workshop '89*, June, Grenoble, France, LNCS 407, pp. 151–165. Springer, Berlin.