Model Checking Applications of Aspects and Superimpositions

Marcelo Sihman and Shmuel Katz
Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel
{sihman, katz}@cs.technion.ac.il

ABSTRACT

The model checking of applications of aspects is explained, by showing the stages and proof obligations when a collection of generic aspects (called a superimposition) is combined with a basic program. We assume that both the basic program and the collection of aspects have their own specifications. The Bandera tool for Java programs is used to generate input for model checkers, although any similar tool could be employed. New *verification aspects* and superimpositions are defined to modularize the proofs, and separate the proof-related code from the program and the aspects. This allows generating and activating a series of model checking tasks automatically each time a superimposition is applied to a basic program, achieving *superimposition validation*. A case study that monitors and checks an underlying bounded buffer program is presented.

1. INTRODUCTION

Aspects help to isolate cross-cutting concerns in programs and designs. Many researchers have been working on programming and design techniques, software evolution and other implications of AOP. However, little work has been done about formal verification of aspects. In this paper, we show in detail how to verify the combination of collections of aspects over basic programs, using model checking techniques. The use of special aspects for verification is also presented, providing yet another natural application of aspect-oriented software design.

We introduce this approach as a new feature of SuperJ, an AOP construct that we have proposed in [15]. SuperJ provides language support for defining collections of parameterized aspects independently of any basic program, where such a collection is called a *superimposition*. A superimposition is a module describing an algorithm that may be applied to different underlying basic programs. A brief introduction to SuperJ is presented in Section 2.

In this paper we consider how model checking of software can be used in the formal verification of combinations of superimpositions and basic programs. Model checking has the advantages of automatic verification (in that difficult invariants do not need to be supplied, as is the case in formal verification based on theorem proving), yet provides full verification, as long as any data abstractions preserve the properties being checked. Additionally, it has proven popular with verification of hardware designs mainly because it provides counter-examples when the property of interest does not hold.

We have chosen Bandera [5] as the prototype generator of input to model checkers such as SMV or Java Pathfinder, and thus use Bandera's specification notation BSL for describing temporal properties to be model checked. A brief introduction to Bandera is given in Section 3.

When binding a collection of aspects (a superimposition) to a basic program (a collection of basic classes), we need to bind each relevant class of the basic program to a generic aspect (of the superimposition), where basic classes may be left unbound to any generic aspect if they do not play a role in the superimposed algorithm.

In a superimposition, we specify *assumptions* about the basic programs and parameters to be bound and *desired results* that must be true in the augmented program, where an *augmented program* is the result after binding a superimposition to a basic program. We assume here that the result of such binding and instantiation (often called *weaving*) is a Java program in itself, rather than, for example, Java byte-code. (The implications of this assumption for our implementation are considered later.) A *superimposition is correct* if, when the aspects in it are woven into a basic program that satisfies the superimposition's assumptions, the augmented program satisfies the desired results and does not violate the original specification of the basic program.

As will be shown, in Bandera, code is added to the program to be model checked in order to define functions, predicates, control locations, and assertions used only for the model checking. We take advantage of the superimposition construct to define *verification aspects* that are used to separate these additions from the code of the programs. All of the verification aspects concerning the assumptions are grouped into an *assumptions superimposition*, and similarly, those related to the results are in a *results superimposition*. The superimpositions and basic programs of the application under consideration can thus be kept free of verification augmentations. This is possible in SuperJ because it supports weaving multiple superimpositions over a basic program, so both the application superimpositions and those needed for verification can be combined before applying Bandera to

generate input for a model checker.

There are several possibilities for using the approach seen here to check superimpositions and their combination with basic programs. These vary according to the modularity in the proof itself, and whether we wish to prove the superimposition correct independently from any specific basic program. In the case of model checking, this may be done by writing a suitable abstraction of a basic program that respects the superimposition requirements, along with an inductive proof. However, we claim here that a more practical alternative is to use the verification superimpositions to set up the automatic generation and activation of four model checking tasks each time a superimposition is applied to a basic program. This procedure, explained and justified later in the paper, is known as *superimposition validation.*

2. SUPERJ

SuperJ introduces constructs that extend the expressiveness and modularity of AOP. Among the new facilities in SuperJ are grouping related aspects into a superimposition, providing specifications, extending parameterization of aspects, dealing with interaction and interference among aspects, and combining superimpositions to obtain new superimpositions. The new superimposition construct comes from the merging of ideas from two distinct research subjects: 'classic' superimposition and AOP.

Well-known examples of 'classic' superimpositions are termination and deadlock detection, monitoring or debugging, adding scheduling restrictions, imposing 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 or *superimpose* an algorithm over a basic program. Numerous suggestions ([1, 2, 3, 4, 10, 11]) 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 [16].

In SuperJ, a superimposition is defined as a collection of generic parameterized aspects and singleton concrete classes. A generic aspect has no built-in connection with any program unit of any basic program, and in contrast to usual aspects, a generic aspect contains an extensive parameter list that allows binding it to any appropriate basic class. The singleton concrete classes define unique objects that must be instantiated in an augmented system, where these unique objects interact with the generic aspects. We have defined an AspectJ-based implementation for SuperJ, and have written a preprocessor that translates SuperJ to pure AspectJ code. The same preprocessor is responsible for several tasks, such as: binding arguments from the basic program (classes, methods, etc.) to the parameters of the generic aspects, and applying a superimposition to a basic program, generating concrete aspects from generic aspects and then weaving them to the basic classes.

3. BANDERA

The Bandera Tool Set [5], as defined by its authors, is an integrated collection of program analysis, transformation, and visualization components designated to allow experimentation with model-checking properties of Java source code. Bandera takes as input an augmented Java source code and a program specification written in Bandera's temporal Specification Language (BSL), and produces a program model and a specification as input to one of four model-checking applications: SMV [12], Spin [8], dSpin [9] and Java PathFinder [7]. This 'input' generated by Bandera is written in the model and specification languages of one of the four model-checking applications mentioned. Then Bandera uses the model-checking application to prove whether the model satisfies the required specification (the Java program satisfies the BSL specification). If the specification is not satisfied, then a counter-example is returned, as is common in model-checking tools. Moreover, Bandera shows the problematic execution path, which does not satisfy the required specification, directly in the Java code.

Bandera deals with the state explosion problem, as the program state model must be finite, by providing data abstraction and program slicing features when customizing the model. These features help produce a much simpler finite-state model of the Java program.

To understand the changes we propose in the verification process, we first need to give a brief introduction to the specification and verification stages in Bandera, and other software model checkers. We ignore some actual limitations imposed by Bandera due to implementation restrictions or arbitrary design decisions not to implement some features of Java, and relate to a somewhat idealized version.

Given a Java program, we need to augment it to include definitions using BSL. For a simple assertion about the state whenever a given location is reached, or pre and post-conditions of a method, we write the assertion definitions - using BSL - as Javadoc comments directly in the source code. An assertion is identified by a @assert tag in BSL, where the three assertion types supported by BSL are identified by the identifiers: LOCATION, PRE and POST.

The specification of a more general temporal program property is divided into defining the predicates to be used in the property's definition, and then separately writing the property itself, using the defined predicates. Predicates are, like simple assertions, also planted directly in the source code, where there are several types of predicates that Bandera allows us to define. For example, we may define a *location* predicate, which is *true* whenever the location is reached (and *false* otherwise), by introducing a Java label at a given control point (inside a method) of the program, and also writing a Javadoc comment (right before the associated method heading) containing the predicate definition in BSL.

An *instance* predicate defines a given property that is not connected to any control point of the program, e.g., invariant properties that must hold during the whole life cycle of an object. In addition, it is also possible to define predicates associated with two different method call control points: when a given method is invoked and when it returns a value. In this case, the predicate evaluates to *true* both when the given method is invoked and when it returns a value. Every predicate definition is written in a Javadoc comment. A predicate definition is identified by a @observable tag in

BSL, where the four predicate types supported by BSL are identified by the identifiers: LOCATION, EXP, INVOKE and RETURN; location, instance, method invoking, and return predicates, respectively.

In the second step needed for defining a given temporal property, after having defined all the predicates that it needs, we need to specify the required temporal property using the temporal specification patterns supported by Bandera, which are: absence, existence, precedence, response and universality. Let P, Q, R be predicates defined using BSL. P is *absent* in a program if it never evaluates to true. P *exists* if it is evaluated to true at least once in the program. P *precedes* Q when P does not evaluate to true before Q is true (which is automatically satisfied when P is *absent*). P *responds* to Q if after Q is true, then P *exists* (which is automatically satisfied when Q is *absent*). P is *universal* if P always evaluates to true.

In Bandera's temporal specification pattern system, we may require a temporal property to hold *globally*, i.e. during all the program execution, or at certain points during the program execution, such as $after\ Q$, $after\ Q\ until\ R$, $before\ Q$, $between\ Q$ and R, where Q and R are predicates defined using BSL.

The temporal specification of a given program is stored in a separate specification file. After having specified all the assertions and temporal properties required for verifying the correctness of the program, we may use Bandera's graphic tool to define a verification session and supply all the data needed, such as the names of the files containing the source code and the specification. When running a correctness check, we may choose exactly which of the assertions and temporal properties defined we want to verify.

Moreover, it is also possible to use data abstractions to simplify the finite-state model generated by Bandera. For example, in a pipeline program shown in [6], a series of integer values, ranging from 1 to 100, is sent from the first stage to the last, passing by all the pipeline stages. When the pipeline program finishes, the first stage sends a 0 value, and then all the stages finish consecutively. In the specification of this example, the integer values - ranging from 1 to 100 - sent in the pipeline are not important. We only need to know when a stage receives a 0 value. Therefore, we may use Bandera's *Signs* data abstraction, which will generate only three different states for the possible values that are sent in the pipeline: negative, zero and positive; instead of more than a hundred different states. Bandera's graphic tool has an interface for defining data abstractions, which we can afterwards store in a separate file. We may also select Bandera's program slicing feature for simplifying the finite-state model generated. After defining the verification session, we only need to run the verification checker, obtaining formal verification of the property if the model checking completes without discovering an error, and otherwise provides a counter-example in terms of the Java code.

In the Appendix, we use a bounded buffer program to give a brief demonstration of all the Bandera concepts introduced in this section. This program is a slightly changed version of an example seen in [6]. Explanations of the example may be also found in the Appendix.

4. PROVING CORRECTNESS IN SUPERJ

4.1 Introduction

In this and the following sections we explain and demonstrate the different options for verifying that a combination of a superimposition and a basic program is correct, as supported by the new features of SuperJ. In Section 4.2, we explain the verification of a combination of a superimposition and a basic program. In Section 4.3, we introduce the intuitively attractive option of proving the correctness of a superimposition independently of any basic program, and discuss the practicality of this option. In Section 5, we use a simple superimposition example to demonstrate some of the concepts introduced by the new SuperJ features, and discuss the implications for superimposition validation in Section 6.

4.2 Superimposition over a Basic Program

In this subsection, we assume a superimposition and a basic program. We want to apply the superimposition over the basic program, checking that the basic program satisfies the superimposition assumptions and that the resulting augmented program is indeed correct, i.e., satisfies all the desired results of the superimposition, as well as the original specification of the basic program. The simplest possibility is to simply view the result of weaving the superimposition's aspects with the basic program as a Java program that should satisfy the original specification, plus the result assertions of the superimposition. Following the description in Section 3, we then may build in all the needed functions, predicates, labels, and BSL statements to the augmented program, create the separate specification file, and model check all at once that the needed temporal BSL assertions are satisfied (or obtain counter-examples).

This is the simplest option for verifying the correctness of a combination of a superimposition over a basic program, since we directly consider the augmented program, and add in all of the needed predicates and assertions in BSL, as seen in the previous section. However, in this case the assumptions and desired results of the superimposition are already instantiated for the combination, and are mixed together with the original specification of the basic program. When a new combination is done, a completely new annotation has to be added before Bandera can be applied. This makes the model checking impractical when the superimpositions are to be used in many contexts. Thus we now propose a better option.

In order to more clearly organize the proofs, and thus to help in identifying the source of any errors, new verification aspects and superimpositions can be used to modularize the treatment. This allows having regular superimpositions and basic programs, free of verification definitions. The extra definitions needed for Bandera's verification are isolated in dedicated aspects, which are used just for proving the correctness of the augmented program in separate steps.

When completely separating the verification definitions from the superimposition and basic program, we have a series of verification aspects that may be sequentially applied to the basic program, or may be combined using combinations of

superimpositions. Moreover, we may now define a *verification superimposition* as a collection of verification aspects. We may classify the verification superimpositions in three different types, defining:

Spec the specification of the basic program;

Asm the superimposition assumptions;

Res the superimposition desired results.

The *Spec* superimposition will have one or more verification aspects, which will contain (AspectJ) advice declarations needed for introducing the verification definitions of the basic program's specification. It also includes the BSL temporal properties which in Bandera are kept in a separate file.

The *Asm* superimposition, dealing with assumptions, will have a collection of verification aspects: one verification aspect for each generic aspect that assumes some properties about the basic class to be bound to it; and one verification aspect for the global assumptions of the superimposition that must be satisfied by the basic program, where these assumptions are not connected to only a generic aspect and its (bound) basic class. Clearly, the assumptions should be as weak as possible, in order to allow applying the superimposition to a large class of basic programs.

A *Res* superimposition is very similar to an *Asm* superimposition, except that it specifies the superimpositions desired results instead of its assumptions. *Res* will also have a collection of verification aspects, like *Asm*.

The complete verification process is composed of four steps:

1. apply *Spec* over the basic program and check its correctness;
2. apply *Asm* over the basic program, and check that the basic program satisfies the superimposition assumptions;
3. apply the superimposition over the basic program, apply *Spec* over the augmented program, and then check that the superimposition does not cancel any desired result of the basic program;
4. apply the superimposition over the basic program, apply *Res* over the augmented program, and check that the augmented program achieves the desired results.

Note that *Spec* is used twice, and that the separation of the verification definitions into aspects and superimpositions is a cleaner solution than the comments used by Bandera to sometimes use and sometimes ignore the verification definitions. Of course, if some of the model checking has already been done for a basic or augmented program, it need not be redone. For example, if the basic program has been shown to satisfy *Spec* once, this need not be redone when applying a superimposition. The parameterization in the verification aspects allows their reuse for different basic programs, with different weavings and instantiations. The advantages of this reuse are further considered in the Discussion section.

4.3 Proving Superimposition Correctness

In this section we consider how to prove that a superimposition is correct independently of any basic program. If we succeed, then we are assured that when this superimposition is applied over a basic program that satisfies its assumptions, then the augmented program will have the superimposition's desired properties. Such a verification is desirable if the superimposition is intended to be put in a library for reuse in many contexts. Of course, if such a proof has been done, we still need only the model checking proofs that the basic program satisfies the assumptions of the superimposition, and that the result of weaving does not violate the specification of the basic program.

The generic correctness requirements and stages in such a proof are not difficult to state in terms of inductive assertions about the structure of every possible basic program to which the superimposition can be applied. However, any such proof has a part which is inductive, and thus non-algorithmic, requiring the invention of inductive assertions. This is true both when the entire proof is based on inductive theorem proving, and when the proof can be divided into a model checking part and an inductive part proving that if the model checking part is successful, then the desired conclusion is justified.

One way to do such a combination of model checking with an inductive proof to obtain a correctness proof of a superimposition uses what can be called *dummy* basic programs, first proposed in [11]. Note that model checking tools verify a model of a fully defined program by checking that the specified properties hold in all execution paths of the program. A superimposition, however, is itself not a program, since it cannot be run, so there are no execution paths. Therefore, we need to write an abstraction of a basic program that fits the superimposition's assumptions, so that we can apply the superimposition over the abstraction. Then we will have execution paths that may be used to prove the correctness of the superimposition combined with the abstract program. This program abstraction may be seen as a dummy basic program.

The dummy program will have no desired results, since it does not do any useful computation. Thus, there will be no *Spec* verification superimposition in the correctness verification process. On the other hand, the other types of verification aspects and superimpositions will still appear, as explained in the previous section. 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, as in the abstraction of message values already mentioned. 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 [14].

Ideally, if the model checking succeeds for the combination of the superimposition over the abstract basic program, then it would succeed for any basic program satisfying the assumptions of the superimposition. However, techniques for proving this ideal conclusion are not yet developed, and in any case they are inductive except when there are trivial structural similarities between the 'real' basic program and the dummy actually model checked. If done successfully, any basic program satisfying the assumptions, and with sufficient components and states to allow binding to the superimposition and its aspects, can be abstracted to this canonic abstract basic program.

In general, the justification that a representative abstraction is indeed sufficient can itself involve infinite or very large state spaces and may require inductive theorem proving. In the Discussion section, we show that by carefully using the techniques in the previous subsection, it may not be necessary to generate such non-algorithmic proof obligations to obtain fully verified combinations of aspects and basic programs in practice.

5. CASE STUDY

5.1 Introduction

In this section, we demonstrate the stages in verifying a combination of a superimposition and a basic program using SuperJ by means of a case study over the Monitoring superimposition, which is shown in Figure 1. Monitoring is a simple superimposition that gathers statistics on basic objects, such as counting the total number of external method calls for all relevant basic objects. The superimposition does not modify the values of the variables. It also checks that objects intended to be constant, actually are - and stops the program when a violation is discovered. It thus does regulate the behavior of the basic program and can affect its properties. In reading the example, note that SuperJ has a keyword BC (an abbreviation for 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.

The Monitoring superimposition contains two generic aspects (Constant and Mutable) and one singleton class (Coordinator). Constant and Mutable extend the Common abstract aspect, which contains code common to both generic aspects. The Common aspect defines the Coordinator class and creates its single instance *coord*, which is used by Constant and Mutable; moreover, Common's advice increments the *nCalls* counter after each external call to any method of the bound class, where each aspect instance will have its particular *nCalls* counter. The Common's allExternalCalls pointcut is defined in both generic aspects of the superimposition (Mutable and Constant). The join points determined by this pointcut - in some bound (basic) object - are all the method calls where the basic object is the callee, but not the caller. In a basic object bound to Mutable, after each field assignment performed, Mutable's advice increments the *nAssigns* counter, where each instance of Mutable has its particular *nAssigns* counter. The only instance of Coordinator (*coord*) accumulates the global statistics gathered by Constant and Mutable. Basic objects intended to be constant, whose field values should not be changed, must be bound to the Constant aspect; and then, if a field assign-

```
superimposition Monitoring {
class Coordinator {
  private int totCalls = 0;
  private int totConCalls = 0;
  private int totMutCalls = 0;
  private int totMutAssigns = 0;

  public void conMethodCount(int x) {
    totConCalls += x; totCalls += x;
  }
  public void mutMethodCount(int x) {
    totMutCalls += x; totCalls += x;
  }
  public void mutAssignCount(int x) {
    totMutAssigns += x;
  }
}

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

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

aspect Constant(EM) extends Common {
  protected pointcut allExternalCalls(): !cflowbelow
    (within(Element)) && execution(* BC.* (..));
  before(): set(* BC.*) &&
            !cflow(initialization(BC.new(..))) {
    System.out.println("Constant err: illegal assignment");
    System.out.exit(-1);
  }
  after(): execution(* BC.EM(..)) {
    coord.conMethodCount(nCalls);
  }
}

aspect Mutable(EM) extends Common {
  protected int nAssigns = 0;

  protected pointcut allExternalCalls(): !cflowbelow
    (within(Element)) && execution(* BC.* (..));
  after(): set(* BC.*) {
    nAssigns++;
  }
  after(): execution(* BC.EM(..)) {
    coord.mutMethodCount(nCalls);
    coord.mutAssignCount(nAssigns);
  }
}
```

Figure 1: A monitoring superimposition.

ment is tried, the aspect prints an error message and finishes the execution of the augmented program.

Each basic object augmented by Mutable will call *coord*'s $mutMethodCount$ and $mutAssignCount$ methods, while objects bound to Constant will call *coord*'s $conMethodCount$ method. The $mutMethodCount$ and $conMethodCount$ methods both update the $totCalls$ common method call counter, and, respectively, update their $totMutCalls$ and $totConCalls$ individual counters. The $mutAssignCount$ method updates the $totMutAssigns$ assignment counter. 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.

The assumptions and desired results of the superimposition are introduced stepwise in Section 5.2, where we verify the correctness of Monitoring over the bounded buffer program (seen in the Appendix), which is used as an example of a basic program.

5.2 Superimposition over a Basic Program

In this subsection we want to apply the Monitoring superimposition over the bounded buffer basic program, and verify the correctness of the augmented program, which we get as a result of their combination. We apply Mutable over BoundedBuffer, binding BoundedBuffer's $finish$ method to Mutable's EM parameter (an abbreviation for End Method). In addition, we apply Constant over Element, binding Element's $finish$ method to Constant's EM parameter. We show the whole verification process stepwise, as introduced in Section 4.

In the first step, we want to check that the basic program itself is correct, i.e., satisfies its specification. In the Appendix, we show the BoundedBuffer class with all the Bandera specification definitions interleaved with its code, where all these definitions are needed for verifying that the basic program satisfies BoundedBuffer's specification when using Bandera. In our approach this is already the result of applying the *Spec* superimposition of the bounded buffer to the original version of the program. This is given as input to Bandera, defining a new verification session with all the information needed by Bandera for running the verification, as shown in Section 3. We then run Bandera's verification to check if all the properties specified are satisfied. In this example, we succeed to show that the basic program is correct, since it indeed satisfies its specification, completing the first stage of the model checking.

In the second step, we want to check that the basic program satisfies all the assumptions specified by the superimposition. For this purpose, we use an *Asm* verification superimposition. *Asm* has a verification aspect for each generic aspect of Monitoring that assumes some property about the basic class to be bound to it. In addition, *Asm* has a verification aspect for the global properties assumed by Monitoring about the basic program, such as invariant properties, which are not connected to only a specific generic aspect.

```
superimposition MonitoringAsm {
  aspect CommonAsm {
   /**
    * @observable
    *     LOCATION[beforeCall] beforeCallLoc;
    *     LOCATION[afterCall] afterCallLoc;
    */
   void around(BC C): target(C) &&
           execution(* BC.*(..)) {
     beforeCallLoc:
     proceed(C);
     afterCallLoc:
   }
  }
  properties {
    alwaysFinishProp: forall [bc:BC].
           {BC.EM.beforeCall(bc)} exists globally;
    singleNoCallAfterFinishProp: forall [bc:BC].
           {BC.*.beforeCall(bc)}
           is absent after {BC.EM.afterCall(bc)};
  }
}
```

Figure 2: Monitoring's *Asm* superimposition

A property that both Mutable and Constant assume about basic classes is that the basic method that is bound to the EM parameter is called exactly once, where the EM parameter must be bound to the last method that is called in the basic object. In the sequel, the basic method bound to EM is called *bound_EM*. Another property that both Mutable and Constant assume is that *bound_EM* is the last method called in every instance of the basic class.

As explained in Section 3, in a usual Bandera verification session we write the specification of the temporal properties to be checked in a separate specification file. However, in SuperJ, we write this specification in a new *properties* section of the verification superimposition. We have written a preprocessor that supports this design decision, which separates the definitions in the properties section from the rest of the superimposition code and then prepares a new verification session for running the verification.

The specification of the properties assumed by the generic aspects need to use two location predicates that must be defined in the basic classes. These two predicates are defined in the verification aspect by the same advice, as shown in Monitoring's *Asm* superimposition, seen in Figure 2.

The single *Asm* verification aspect must be applied over all the basic classes to be bound to Constant and Mutable. In the bounded buffer example, they are applied, in turn, over Element and BoundedBuffer. The two predicates defined in the verification aspects are associated with two locations in each method of every basic class bound to Constant or Mutable (e.g. Element and BoundedBuffer). Each of these predicates is true during execution when the augmented program reaches the control points where they were defined, i.e., in an execution path. The control points associated with these predicates (beforeCall and afterCall) are right before the first and after the last commands executed

in the basic methods of Element and BoundedBuffer.

After having defined the two predicates needed for the verification, we can write the two properties that, if satisfied, will ensure that the basic program satisfies the two assumptions, which are required by both Mutable and Constant. These two properties are written in temporal logic using BSL, and appear in the properties section of the *Asm* superimposition.

The first property, which is called alwaysFinishProp, checks that *bound_EM* is eventually called. However, that is not enough, since we want this method to be called exactly once, and no other method to be called after that. Therefore, the second property (singleNoCallAfterFinishProp) checks that no basic method will be called after *bound_EM* is called.

We put a '*' character in the place where we should write the name of the basic method where beforeCall was defined. The '*' character fits every method of the basic class. Unfortunately, BSL does not support this special '*' character. In a usual Bandera specification, we need to write separate temporal properties for each method of the basic class. However, our preprocessor overcomes this limitation, generating all the properties needed for every method of the basic class.

In the example seen, both Mutable and Constant shared exactly the same requirements, so in this particular case we can use the same *Asm* aspect for both generic aspects. However, if the assumptions required by two distinct generic aspects differ, then we obviously need to write them in two separate aspects. Moreover, Monitoring does not assume any global property about the basic program, so there is no *Asm* aspect for checking if the global assumptions are satisfied.

At this stage, we are able to apply the verification superimposition over the basic program. We then create a new verification session for checking the superimposition assumptions, and then run the verification in Bandera.

After having demonstrated the second step of the new verification feature, we now go on to the third step, where we check that the superimposition does not cancel any of the desired results of the basic program. Initially, we need to apply the superimposition over the basic program, e.g., Mutable over BoundedBuffer and Constant over Element. Finally, we apply the *Spec* verification superimposition - containing the verification definitions needed for checking the basic program's specification - over the augmented program. Here we do not show the *Spec* superimposition, since we show its verification definitions interleaved with the code of the bounded buffer in the Appendix, together with its specification file. We then supply all the data that Bandera needs for the desired check and run the verification. If the augmented program passes the verification, then we are assured that the superimposition does not cancel any desired result of the basic program.

In the fourth and last step of the verification process, we want to check that the augmented program has all the desired results specified by the superimposition. For this purpose, we apply the superimposition over the basic program (Monitoring over the bounded buffer program), and then

```
superimposition MonitoringRes {
 /**
  * @observable
  * EXP Eq: (totCalls == (totConCalls + totMutCalls));
  */
 aspect GlobalRes {
  /**
   * @observable
   *     LOCATION[beforeConMC] beforeConMCLoc;
   *     LOCATION[afterConMC] afterConMCLoc;
   */
  void around(Coordinator C): target(C) &&
     execution(void Coordinator.conMethodCount(int)) {
    beforeConMCLoc:
    proceed(C);
    afterConMCLoc:
  }
  /**
   * @observable
   *     LOCATION[beforeMutMC] beforeMutMCLoc;
   *     LOCATION[afterMutMC] afterMutMCLoc;
   */
  void around(Coordinator C): target(C) &&
     execution(void Coordinator.mutMethodCount(int)) {
     beforeMutMCLoc:
     proceed(C);
     afterMutMCLoc:
  }
 }

 aspect ConstantRes {
 /**
  * @observable
  * LOCATION[beforeConFieldSet] beforeConFieldSetLoc;
  * LOCATION[afterConFieldSet] afterConFieldSetLoc;
  */
 before(): set(* Element.*) &&
         !cflow(initialization(Element.new(..))) {
  beforeConFieldSetLoc:
 }
 after(): set(* Element.*) &&
        !cflow(initialization(Element.new(..))) {
  afterConFieldSetLoc:
 }
 properties {
  totCallsEqBeforeProp: forall [c:Coordinator].
   {Eq(c)} is universal
   before{Coordinator.conMethodCount.beforeConMC(c) ||
          Coordinator.mutMethodCount.beforeMutMC(c)};
  totCallsEqAfterProp: forall [c:Coordinator].
   {Eq(c)} is universal
   after{Coordinator.conMethodCount.afterConMC(c) ||
         Coordinator.mutMethodCount.afterMutMC(c)};
   until{Coordinator.conMethodCount.beforeConMC(c) ||
         Coordinator.mutMethodCount.beforeMutMC(c)};
  conObjTermIfSetProp: forall [bc:BC].
   {BC.*.afterConFieldSetLoc(bc)}
   is absent after {BC.*.beforeConFieldSetLoc(bc)};
 }
}
```

Figure 3: Monitoring's *Res* superimposition

we apply the *Res* superimposition over the augmented program, where the *Res* superimposition checks that all the desired results of Monitoring are present in the augmented program. The complete *Res* verification superimposition is shown in Figure 3.

A desired result that the superimposition requires the augmented program to satisfy is that the value of Coordinator's field *totCalls* must be always equal to the value of its *totConCalls* field plus the value of its *totMutCalls* field, except when the augmented program is executing one of the two methods of Coordinator that change the values of these fields (*conMethodCount* and *mutMethodCount*). We need to define four predicates in the Coordinator class before specifying the required property in BSL. In addition, an instance predicate must be also defined, stating the desired result itself. Thus, the *Res* aspect associated with Monitoring must contain the definition of the instance predicate, and two advice declarations defining the other four predicates needed.

Moreover, Constant has one desired result and Mutant has none. The desired result of Constant is that the augmented program terminates if a field assignment is tried in the basic object bound to Constant. Therefore, we must write a *Res* aspect associated with Constant. Mutable does not need a separate *Res* aspect beyond the global required result of the Monitoring superimposition.

The four predicates - defined by the global *Res* aspect - are associated with the augmented program's control points before and after the *conMethodCount* and *mutMethodCount* method, respectively. The *Eq* instance predicate defines the property that must be satisfied in the augmented program. The two predicates - defined by the *Res* aspect associated with Constant - will be true before and after a field assignment is tried in the basic object bound to Constant (an instance of Element).

We write the specification of the superimposition desired results, using BSL, in the properties section. In the two first properties seen, we specify that the *Eq* property must hold from the beginning of the execution of the augmented program until either *conMethodCount* or *mutMethodCount* is called, and after finishing to execute either one of them until calling one of them again. The third temporal property specifies that if the augmented object bound to Constant (an instance of Element) reaches the control point right before a field assignment, then it will not reach the control point right after the field assignment.

Above, we have seen a demonstration of the complete process of verifying the correctness of a superimposition over a basic program. The augmented program that we get from applying Monitoring over the bounded buffer program passes all the stages of the verification process. However, some slight changes in the bounded buffer program could cause it to not satisfy the assumptions required. For example, if we substitute an infinite loop in place of the *for* loops of InOut1 or InOut2 that take and add an element from the buffer one hundred times, the model checking produces a counter-example and shows incorrectness. This is because the *finish* methods of the buffer and its elements would never be called, violating one of the assumptions of the Monitoring superimposition.

If the Monitoring superimposition could change the indices of the buffer of the underlying bounded buffer program, a counter-example would be produced when *Spec* were model checked for the augmented program (in stage 3), because the assertions involving the indices would be violated.

6. DISCUSSION: SUPERIMPOSITION VALIDATION

The separation of verification annotations into the different verification superimpositions described above allows a clean application of instances of model checking for combinations of superimpositions and basic programs. Note that when a verification superimposition is woven either with a new basic program or with the augmented program obtained after weaving the application superimposition and the basic program, the weaving process binds classes, methods, fields, and pointcuts of the generic verification superimposition to those of the application. No change is needed in *Asm* or *Res* themselves. Of course, the specification of the new basic program, *Spec* needs to be produced, and expressed as a verification superimposition.

Once the bindings have been determined, the entire process is in principle automatic, ignoring practical restrictions of the tools involved. When a superimposition is woven with a basic program, SuperJ's preprocessor generates AspectJ code, and AspectJ's preprocessor is used in the mode which generates source Java code. Then for each of the four steps described, the appropriate verification superimposition is woven with the basic or augmented program, as appropriate, and the processing of SuperJ and AspectJ are again activated, to obtain 'Bandera-ready' Java. Bandera then is applied to generate input to a model checker such as SMV, and the algorithmic model-checking either succeeds in verifying or generates a counter-example.

Therefore, although it might seem expensive to model check every combination of a superimposition with a basic program, this is in fact a viable alternative to the inductive (non-algorithmic and therefore very difficult) proof that a superimposition is always correct. The time-consuming, and difficult manual creation of the BSL annotations only needs to be done once for each superimposition, even though the model checker is used for each combination.

Such an alternative is analogous to the idea of *translation validation*, first seen in [13], where assertions are generated and automatically checked whenever a compiler is applied to a source program. The correctness of the assertions implies that *for this activation* the translation of the compiler is correct. This is instead of a full verification of the correctness of the compiler, which is too difficult for non-toy compilers. As here, the key to its practicality is that the generation and verification of the needed assertions is completely automatic for each compilation, and only takes seconds to perform. Similar ideas are seen in some versions of proof-carrying code, that show there are no memory leaks for a particular instance of an applet.

In this paper we have shown how *superimposition validation*

can be similarly applied whenever an application superimposition is woven, if the needed verification superimpositions have been prepared. The other alternative - of a full correctness proof for a superimposition - is, of course still a desirable research goal. However, due to the inductive proof involved, doubt remains that such results can be applied in practice. In any case, the direction seen here does provide the first pathway to practical machine proofs for combinations of aspects and superimpositions with basic programs.

7. REFERENCES

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

[2] L. Bougé and N. Francez. A compositional approach to superimposition. In *ACM Symposium on Principles of Programming Languages*, pages 240–249, Jan 1988.

[3] K. Chandy and J. Misra. *Parallel Program Design - a Foundation*. Addison-Wesley, 1988.

[4] N. Francez and I. Forman. *Interacting Processes*. Addison-Wesley, 1996.

[5] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *CONCUR 2001*, LNCS 2154, pages 39–58, Aug 2001.

[6] J. Hatcliff and O. Tkachuk. The Bandera tools for model-checking Java source code: A user's manual. Technical report, Kansas State University, Department of Computing and Information Sciences, March 2001. `http://www.cis.ksu.edu/%7Esantos/bandera/tut/tut-html.tar.gz`.

[7] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), Apr 2000.

[8] G. J. Holzmann and M. H. Smith. The model checker SPIN. *IEEE trans. SE*, 23(5):279–295, 1997.

[9] R. Iosif and R. Sisto. dspin: A dynamic extension of spin. In *Proc. of the 6th SPIN Workshop*, LNCS 1680, pages 261–276, Sep 1999.

[10] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings ICSE'90*, pages 63–71. IEEE Press, 1990. `http://disco.cs.tut.fi`.

[11] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. on Programming Languages and Systems*, 15(2):337–356, Apr 1993.

[12] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[13] A. Pnueli, O.Shtrichman, and M.Siegel. The code validation tool(cvt) - automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2:192–201, 1999.

[14] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Proc. of the international workshop on Automatic verification methods for finite state systems*, pages 151–165. Springer-Verlag, 1990.

[15] M. Sihman and S. Katz. Superimposition and aspect-oriented programming. to appear in BCS Computer Journal. Available at `http://www.cs.technion.ac.il/~katz/cj.ps`.

[16] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *Proceedings of AOSD 2002*, pages 28–40. ACM Press, Apr 2002.

APPENDIX

A. BOUNDED BUFFER EXAMPLE

A.1 Introduction

The bounded buffer example is a multi-threaded Java program introduced in [6] as an example for demonstrating a verification session in Bandera. The BoundedBuffer class has three methods: $add(Element)$, $take()$, $isEmpty()$. When the buffer is not full, the *add* method adds an Element object to the buffer, which is defined as a fixed array of Element objects. The *take* method takes an Element object (*element*) from the buffer, if the last is not empty. The *isEmpty* method returns true when the buffer is empty, and false otherwise. The constructor of BoundedBuffer receives (as parameters) the size of the buffer array and the number of threads running (using the bounded buffer), and then initializes all the object fields.

The other classes that appear in this example are: CompleteBoundedBuffer, InOut1, InOut2 and Element. The first is the main driver class that runs the program, creating two BoundedBuffer instances and single instances of InOut1 and InOut2. The InOut1 instance is a thread that contains a finite loop where it takes an *element* from the first buffer and adds it to the second, while the InOut2 instance has an identical finite loop that takes an *element* from the second buffer and adds it to the first buffer. CompleteBoundedBuffer creates two *elements* and adds them respectively to the first and second buffers, where an *element* contains an *Object* instance as its only field, and has two methods that allow changing and getting the *Object* instance that it contains. Both BoundedBuffer and Element classes contain a $finish$ method that performs computation destined to be executed when the program finishes.

Five properties are checked in the bounded buffer example. The BoundedBuffer's constructor parameter (for the size of its array) must be a positive number, which is specified by the PositiveBound assertion. The *add* method always adds the *element* in correct position, which is specified by the addPost assertion. The buffer indices ($head\_$ and $tail\_$ BoundedBuffer fields) always stay in range, which is specified by the temporal property IndexRange, which uses the IndexRange instance predicate. A full buffer eventually becomes non-full, which is specified by the FullToNonFull temporal property, which uses the Full instance predicate. An empty buffer must have an *element* added to it before an *element* is taken from it, which is specified by

the NoTakeWhileEmpty temporal property, which uses the Empty instance predicate and the takeReturn and addInvoke location-sensitive predicates.

A.2 Source Code

```
public class CompleteBoundedBuffer {
 public static void main (String [] args) {
    BoundedBuffer b1 = new BoundedBuffer(3,2);
    BoundedBuffer b2 = new BoundedBuffer(3,2);
    b1.add(new Element(new String("1")));
    b2.add(new Element(new String("2")));
    (new InOut1(b1,b2)).start();
    (new InOut2(b2,b1)).start();
 }
}

class Element {
 Object obj;
 Element(Object o) {···}
 public void set(Object o) {···}
 public Object get() {···}
 public void finish() {···}
}

/**
 * @observable
 * EXP Full: (head_ == tail_);
 * EXP Empty: head_ == ((tail_+1) % bound_);
 * EXP IndexRange: (head_ >= 0 && tail_ >= 0 &&
 *                  head_ < bound_ && tail_ < bound_);
 */
class BoundedBuffer {
 Element [] buffer_;
 int bound_;
 int head_, tail_;
 int nThreadsRun, nThreadsEnd = 0;

/**
 * @assert
 * PRE PositiveBound: (b > 0);
 */
 public BoundedBuffer(int b, int n) {···}

/**
 * @assert
 * POST addPost: (head_==0) ? buffer_[bound_-1]==o :
 *                            buffer_[head_-1]==o;
 * @observable
 * INVOKE addInvoke;
 */
 public synchronized void add(Element o) {···}

/**
 * @observable
 * RETURN takeReturn;
 */
 public synchronized Element take() {
    ...
    successTake:
    ...
 }

 public synchronized boolean isEmpty() {···}

 public synchronized void threadFinished() {
    if (++nThreadsEnd == nThreadsRun) {
       finish();
    }
 }

 public synchronized void finish() {···}
}

class InOut1 extends Thread {
 BoundedBuffer in_,out_;
 public InOut1(BoundedBuffer in, BoundedBuffer out) {···}
 public void run() {
    ...
    for(int i=0; i<100; i++) {···}
    in_.threadFinished();
    out_.threadFinished();
 }
}

class InOut2 extends Thread {
 BoundedBuffer in_,out_;
 public InOut2(BoundedBuffer in, BoundedBuffer out) {···}
 public void run() {
    ...
    for(int i=0; i<100; i++) {···}
    in_.threadFinished();
    out_.threadFinished();
 }
}
```

A.3 Specification

```
PositiveBoundAndPost: enable assertions
      {PositiveBound, addPost};

IndexRange: forall[b:BoundedBuffer].
      {IndexRange(b)} is universal globally;

FullToNonFull: forall[b:BoundedBuffer].
      {!Full(b)} responds to {Full(b)} globally;

NoTakeWhileEmpty: forall[b:BoundedBuffer].
      {BoundedBuffer.take.takeReturn(b)} is absent
      after {BoundedBuffer.Empty(b)}
      until {BoundedBuffer.add.addInvoke(b)};
```