Aspects and Superimpositions (Position Paper)

Shmuel Katz Joseph (Yossi) Gil
Computer Science Department
The Technion, Haifa 32000, Israel
{yogi,katz}@cs.technion.ac.il

April 16, 1999

The position advocated by this paper is that the aspect-oriented programming (AOP) community should become familiar with and exploit an existing body of research in distributed systems that explores a notion called *superimposition*. Just as in AOP, superimposition is orthogonal to the usual breakdown into modules. Superimposition makes it possible to break the design of a system into several subtasks. One subtask is to be solved by the basic algorithm that the system has to implement. Other subtasks could insure liveness (progress of the computation), increase parallelism, take snapshots, etc. A superimposer then *interleaves* these separate designs into a single program operating in the same state space.

Superimposition is very natural to distributed systems, where maintaining certain properties of the computation is both important and difficult. Examples are modules for additional monitoring of existing processes, debugging aids, repeated checking of whether deadlock has occurred, increasing fault tolerance, or introducing buffers to reduce the synchronization among processes. Indeed, algorithms which were intentionally designed to superimpose additional functionality on a basic program have a long history in distributed algorithms research, probably starting with algorithms to detect termination of basic algorithms (e.g., [5, 6, 8]).

In those algorithms, new actions described as "pieces of code" are interspersed with the regular actions in a system to ensure proper termination of a system that otherwise would deadlock with each process waiting indefinitely to serve the others. Another well-known example is the global snapshot algorithm of Chandy and Lamport [3], that takes a snapshot of a distributed system while the underlying computation continues, interleaved with the actions of the underlying system.

Based on these and other examples, several versions of a language construct called a *superimposition* have been proposed, either independently of a specific language [9, 2] or as part of a notation for distributed programming [4, 7, 1]. They are intended to allow easy expression of modules that add functionality to a distributed system but require access to the state of the existing processes and execute interleaved with the rest of the computation, augmenting the original system at many different places.

The motivation of superimpositions is identical to that seen in the aspects approach to object oriented programming. They both cut across the "normal" structuring of systems (into objects and classes for object systems, and into processes and groups of processes connected by channels for distributed systems). They both seek to add functionality to a system without disturbing the desirable properties of the original system. Thus it is natural to consider to what degree the research on superimpositions has implications on the future development of aspects.

We identify at least the following dimensions in which AOP can benefit from the maturity of the research on superimposition:

Syntax and language design

Taxonomy of increasing expressiveness/intervention

Modular specification

Verification and application of formal methods

In particular we consider proposals for language constructs to support superimposition, and the criteria for specifying and showing a superimposition correct independently of a particular basic algorithm. These proposals were intended to allow reuse of superimpositions, and should be equally applicable to aspects.

The syntax of superimpositions provides both for *declarations* and for a *combination* or binding operator to produce augmented systems (and which has obvious connections with the *weaving* of AOP). The declarations, which have formal parameters as well as local variables and new processes, use syntactic objects called *roletypes* to define types of augmentations. Several suggestions have been made for a syntax that allows augmenting basic processes in ways that aspects have not allowed so far. The combination operator, with actual parameters, connects processes of the basic (underlying) system to roletypes.

Additional implications of superimpositions for aspects can be seen in the classification of superimpositions into three types seen in [9]: *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 program without the superimposition.

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.

We also show how a superimposition can be separately declared, specified, and verified. The specification involves, as it will for aspects, (1) the applicability conditions about the basic systems with which it can be combined, (2) the added functionality it is guaranteed to provide, and (3) the families of properties it will maintain. Assuming the basic system satisfies the applicability conditions, both the properties the combination is guaranteed to maintain, and those it adds to the functionality of the underlying system can be proven once and for all. This can be done using a generic combination of the superimposition with an abstracted version of a basic system about which we know only the applicability conditions— namely, the minimum necessary to guarantee the required properties of the combination. Even though a superimposition (like an aspect) usually cannot meaningfully execute 'on its own' without being added to an underlying system, the combination with the generic abstraction, called a *dummy* system, can be formally verified. Then a successful combination with a concrete system is guaranteed if that system can be shown to satisfy the applicability conditions, without reproving other parts of the specification. Again, the use of this technique is valuable for verifying aspects, and for using specified and verified aspects as the basis for establishing an aspect library.

Just as early work on superimpositions took a macro-like code implementation view of their meaning, aspects have been described in this way. However, it was found more effective to view superimpositions as separate entities, with various ways of combining them with basic systems and with each other, using a binding operator. Treating aspects in this way will also provide a firmer theoretical basis and clearer semantics for this important modularity concept.

References

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

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

[3] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, Feb 1985.

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

[5] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11:1–4, Aug 1980.

[6] N. Francez. Distributed termination. *ACM Trans. Prog. Lang. Syst.*, 2:42–55, Jan 1980.

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

[8] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *IEEE Trans. on Software Eng.*, 8(3):287–292, May 1982.

[9] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. on Programming Languages and Systems*, 15:337–356, April 1993.