always be within bounds, but we cannot reasonably expect an automated tool for extracting control flow graphs to perform such inferences. Whether to include some or all implicit control flow edges in a CFG representation therefore involves a trade-off between possibly omitting some execution paths or representing many spurious paths. Which is preferable depends on the uses to which the CFG representation will be put.

Even the representation of explicit control flow may differ depending on the uses to which a model is put. In Figure 5.3, the for statement has been broken into its constituent parts (initialization, comparison, and increment for next iteration), each of which appears at a different point in the control flow. For some kinds of analysis, this breakdown would serve no useful purpose. Similarly, a complex conditional expression in Java or C is executed by "short-circuit" evaluation, so the single expression $i > 0$ && $i < 10$ can be broken across two basic blocks (the second test is not executed if the first evaluates to false). If this fine level of execution detail is not relevant to an analysis, we may choose to ignore short-circuit evaluation and treat the entire conditional expression as if it were fully evaluated.

5.4 Call Graphs

The intraprocedural control flow graph represents possible execution paths through a single procedure or method. Interprocedural control flow can also be represented as a directed graph. The most basic model is the *call graph*, in which nodes represent procedures (methods, C functions, etc.) and edges represent the "calls" relation. For example, a call graph representation of the program that includes the collapseNewlines method above would include a node for StringUtils.collapseNewlines with a directed edge to method String.charAt.

Call graph representations present many more design issues and trade-offs than intraprocedural control flow graphs; consequently, there are many variations on the basic call graph representation. For example, consider that in object-oriented languages, method calls are typically made through object references and may be bound to methods in different subclasses depending on the current binding of the object. A call graph for programs in an object-oriented language might therefore represent the *calls* relation to each of the possible methods to which a call might be dynamically bound. More often, the call graph will explicitly represent only a call to the method in the declared class of an object, but it will be part of a richer representation that includes inheritance relations. Constructing an abstract model of executions in the course of analysis will involve interpreting this richer structure.

Figure 5.6 illustrates overestimation of the *calls* relation due to dynamic dispatch. The static call graph includes calls through dynamic bindings that never occur in execution. The call graph includes an (impossible) call from A.check() to C.foo() because A.foo() calls myC.foo() and myC's declared class is C. However, since myC is always an object of subclass S, and S overrides foo(), the call to myC.foo() can only reach S.foo(). In this case a more precise analysis could show that myC is always bound to an object of subclass S, but in general such precision is expensive or even impossible.

If a call graph model represents different behaviors of a procedure depending on where the procedure is called, we call it *context-sensitive*. For example, a context-