Knowledge Compilation: Bridging the Gap between Specification and Implementation

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, OR 97331
tgd@cs.orst.edu

The fundamental goal of artificial intelligence is to develop methodologies for constructing effective, intelligent systems. At the most abstract level, there are three basic subproblems to be solved: (a) *specifying* the desired behavior of the target system, (b) constructing a (run-time) problem-solving architecture that can support effective performance, and (c) bridging the gap (assuming one exists) between the specifications and the run-time architecture.

Researchers in "knowledge compilation" claim that the gap between specifications and run-time architectures is substantial, and they seek automatic methods for transforming specifications into implementations.

Let us examine this claim. The great majority of expert systems make no separation between specifications and run-time implementations. Indeed, one of the major selling points of early expert systems work was that the "specifications" could be gathered incrementally, expressed transparently as rules, and executed interpretively to yield effective problem solving and useful explanations of system reasoning. Clearly such systems contain no gap and require no knowledge compilation.

As we move to more ambitious AI applications, however, the gap begins to appear. Three forces act to open the gap. First, in some domains it is easiest to specify the desired performance of the system in a form that is *too inefficient* to execute interpretively. Second, there are domains where it is *impossible* to interpretively execute the specifications, because they are expressed in terms of unobservable quantities. Third, specifications may contain contradictions or conflicting criteria that must be detected and resolved prior to execution.

As an example of inefficient specifications, consider tasks in engineering design. In design domains, the knowledge taught in textbooks and easily incorporated into computer systems is *analysis* knowledge—knowledge that is sufficient to determine whether a proposed design will satisfy its requirements. Virtually all computer-aided engineering in mechanical design, for example, involves finite-element analysis of structures, heat flow, fluid flow, and so on. A common criticism of modern engineering education is that the only methods taught to students are analysis methods, and consequently newly graduated engineers have difficulty designing successful products.

It is possible, in principle, to apply analysis knowledge directly to perform design tasks. All one needs is an exhaustive generator of possible designs. Then the analysis knowledge can be employed to test these designs until a satisfactory one is found. Of course such an approach is far too inefficient to be practical.

Consequently, rather than employing generate-and-test methods, most design systems are built around a generate-analyze-patch architecture. The process begins by applying *synthesis rules* that map from the requirements directly to a proposed design. This design is then analyzed to determine which requirements are violated. Then a set of *patching rules* can be applied to map from violations to design changes.

In principle, synthesis and patching rules can be derived from analysis knowledge, but the gap is substantial. Knowledge compilation methods can bridge this gap. For example, the LEAP system for VLSI design[1] incrementally compiles analysis knowledge (sufficient to verify the correctness of a design) into synthesis rules.

Because of the large gap between analysis and synthesis in design systems, I believe that knowledge compilation methods will have their greatest impact in engineering design domains.

The second problem—that the specifications may be unexecutable—was first pointed out by Mostow in his system for playing Hearts[2]. A typical specification of the goal of Hearts is to "avoid taking points"—but this specification cannot be satisfied, because the only way to avoid taking points is to know what cards are in your opponents' hands. Of course, after a game is finished, this information is available. However, during the game, a program must find observable "proxy" variables that can predict what cards are in an opponent's hand. For example, if an opponent is unable to follow suit, this reveals that the opponent has no cards in that suit (i.e., the opponent is "void"). Mostow's system therefore adopts the strategy "Lead cards in suits that where opponents are not void."

As another example, consider the problem of controlling a machine tool. Part of the specification might be "do not allow the temperature of the drill bit to exceed 400 degrees." However, typically it is impossible to measure the temperature of the bit directly. Instead, a strategy such as "do not operate the machine continuously for more than 150 seconds" must be adopted.

The third problem—conflicting specifications—arises quite frequently, particularly in design and configuration domains. For example, in the MYCIN therapy selection algorithm, the specifications include "minimize the number of drugs prescribed" and "maximize the number of suspected organisms covered." Mostow and Swartout[3] describe a collection of strategies for resolving these conflicts. Specifically, the minimize drugs/maximize coverage conflict can be resolved by lexicographical ordering. First, we insist on covering the most likely organisms. Then we consider all drug therapies that achieve this and choose one that minimizes the number of drugs. In choosing a strategy to apply, a system must consider both the appropriateness of the strategy and also its efficiency. To support this, each strategy also has an attached implementation.

Given that there can be a gap between the specifications and the run-time architecture, there are two general approaches to closing this gap. One approach, which has been the focus of knowledge compilation research, is to convert the specifications into a form that can be interpreted directly by the run-time architecture. The other approach, which has been the focus of work on model-directed reasoning and task-specific architectures, is to change the run-time architecture so that it can interpret directly the given specifications.

Figure 1 provides a schematic view of these approaches. In the figure, circles represent agents—that is, people or intelligent programs—and squares represent specifications. In Figure 1(a), agent A_X (the "expert") provides specification S_0, but this specification cannot be interpreted directly by agent A_{IE} (the "inference engine"). This illustrates the gap.

Figure 1(b) shows the knowledge compilation approach. Agent A_C (the "compiler") transforms S_0 into S_C, the compiled specification, which is then interpreted directly. One claim of knowledge compilation researchers is that, in current expert systems, agent A_C is often the knowledge engineer. This may provide part of the explanation for why knowledge engineering is so time-consuming. To my knowledge, no one has performed the studies of real systems that are needed to support or refute this claim.

Finally, Figure 1(c) shows the interpreter-modification approach. Agent A_C constructs a specialized interpreter, A_{new}, that is able to interpret the given specification S_0 directly. Obviously, cases (b) and (c) could be combined so that A_C reformulates the specification *and* produces an interpreter for that specification.

When feasible, the interpreter-modification approach is probably best, because interpreters can easily provide support for explanation, incremental modification, and source-level debugging. However, there are many domains where I believe the interpreter-modification approach will not work, and the examples given above—engineering design, unexecutable specifications, and contradictory specifications—demonstrate this. Interpreters are necessarily local in their focus. Hence, they have great difficulty working with the kinds of global information available from engineering analysis knowledge. Similarly, the process of formulating proxy variables for unobservable quantities typically involves lengthy, global analysis. Finally, interpreters cannot perform the kind of global reasoning required to detect and resolve contradictory specifications. From these arguments, we must conclude that in many domains, the knowledge compilation approach is the only feasible strategy.

Given that we wish to build a knowledge compilation system, how can we go about designing and implementing it? There is no single "knowledge compilation method" that will solve our problem. Rather, a large collection of methods have been explored for aiding knowledge compilation and many more methods need to be developed and studied.

To make problem solving more efficient, methods such as caching of inferences, explanation-based learning, partial evaluation of the problem solver, (admissable) heuristic search, and introduction of abstractions can be applied (see, Mitchell, Keller, and Kedar-Cabelli[4], Kahn[5], Mostow and Prieditis[6], Knoblock[7]). All of these methods maintain the correctness of the target system. In addition, it is sometimes necessary to sacrifice correctness to obtain efficiency. For example, it is often convenient to specify design tasks in terms of minimizing cost while maximizing performance parameters. In practice, however, satisficing methods[8] are applied to find a design of *sufficiently* low cost and *sufficiently* high performance.

To address the problem of unexecutable specifications, one approach is to analyze the domain to find observable proxy variables that can approximate unobservable quantities. Another approach is to adopt conservative strategies that are guaranteed not to violate the specifications.

Finally, to resolve the problems of conflicting specifications, strategies of the sort formalized by Mostow and Swartout[3] can be employed.

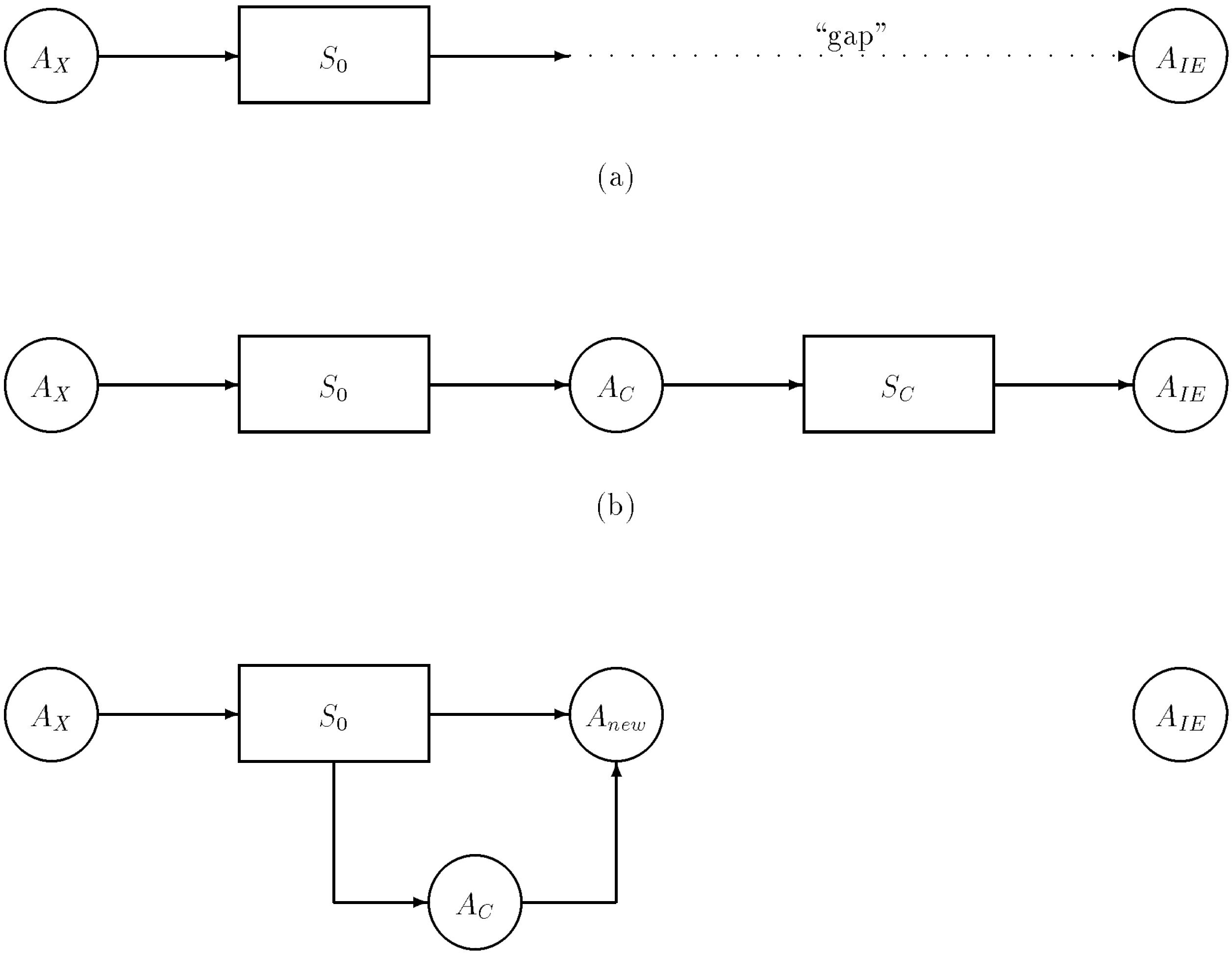

Figure 1: Strategies for Bridging the Gap

This concludes this review of the need for knowledge compilation and the methods available. Let us now consider the implications of this review for some of the remaining issues raised by Ashok Goel.

One issue is the question of how knowledge compilation relates to the deep/shallow knowledge distinction. My response is that *if* the expert finds it easiest to provide specifications in the form of "deep" knowledge and *if* the run-time architecture is only effective with "shallow" rules, then the knowledge compilation process is indeed the process of converting from deep to shallow knowledge. However, the work of Davis and others on model-based reasoning suggests that it is possible in some domains to build a run-time architecture for diagnostic tasks that can interpretively evaluate the "deep" knowledge with reasonable efficiency. In such domains, the knowledge compilation approach is not needed.

In domains where experts have many years of experience, it is unlikely that they will express their knowledge in "deep" form. As many knowledge engineers have discovered, experts are more likely to describe their expertise in the form of associational rules. Consequently, knowledge compilation may have little utility in such domains.

On the other hand, in domains involving new technologies, the only knowledge initially available tends to be "deep." This reflects the fact that it takes 5–10 years to become an expert in a field[9]. For these domains, knowledge compilation may be able to provide a crucial speedup: 5–10 years of human experience can perhaps be compressed into 5–10 months of CPU time. In these kinds of domains, it is appropriate to view the knowledge compilation process as automatically deriving the kinds of "expertise" that experts acquire over many years. Lenat's[10] work on automatically discovering design heuristics for 3-dimensional VLSI technology is a nice example of this.

Another issue raised by Goel is the question of *when* it is worthwhile to perform knowledge compilation. The arguments above would indicate that knowledge compilation should be performed only when it is essential for producing effective performance. However, some would argue that there are additional benefits to be gained from specifying an expert system using "deep knowledge" and then compiling it to produce a run-time system, even if the deep knowledge is harder to acquire from domain experts.

The main benefit is that the final system is more likely to be complete and correct than is a system based on associational rules obtained directly from human experts. Another benefit is that deep knowledge systems give us a better understanding of the problems and tradeoffs in the domain. For example, our understanding of medical diagnosis in general, and of MYCIN in particular, is much better because of Clancey's efforts to "de-compile" expert-supplied rules to make explicit the underlying knowledge and strategies[11].

Let me conclude with a summary of the main points: Knowledge compilation is the process of automatically bridging the gap between a set of specifications (easily provided by the domain expert) and a run-time problem-solving architecture. This gap can arise because the specifications are incomplete, impossible to execute, or contradictory. There are two approaches to closing the gap: knowledge compilation—which reformulates the specifications to match the architecture—and architecture modification—which modifies the architecture to match the specifications. Each approach is appropriate in some domains but not in others. There are many different knowledge compilation methods, and research is still at an early stage. Knowledge compilation methods hold promise of expanding the range of systems that can be constructed and deployed, particularly in areas such as engineering design.

Acknowledgments

The author thanks Jim Bennett for many discussions of this topic and for providing comments on drafts of this paper. This research was supported by NASA Ames Research Center under grant number NAG 2-630.

References

[1] Mitchell, T.M., Mahadevan, S., and Steinberg, L. I. LEAP: A learning apprentice for VLSI design. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 573–580, Los Angeles, CA, 1985. Morgan Kaufmann.

[2] Mostow, D. J. Machine transformation of advice into a heuristic search procedure. In R. S. Michalski, J. Carbonell, and T. Mitchell, editor, *Machine Learning: an artificial intelligence perspective*, volume 1, pages 367–404. Morgan Kaufmann, San Mateo, CA, 1983.

[3] Mostow, D. J. and Swartout, B. Towards explicit integration of knowledge in expert systems: An analysis of MYINC's therapy selection algorithm. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, volume 2, pages 928–935, San Mateo, CA, 1986. Morgan Kaufmann.

[4] Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[5] Kenneth M. Kahn. Partial evaluation, programming methodology, and artificial intelligence. *AI Magazine*, 5(1):53–57, 1984.

[6] Mostow, J., and Prieditis, A. Discovering admissible heuristics by abstracting and optimizing: a transformational approach. In *Proceedings of the Eleventh International Joint Conference on Artficial Intelligence*, pages 701–707, Detroit, MI, 1989. Morgan Kaufmann.

[7] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, Boston, MA, 1990. AAAI Press.

[8] Ellman, T. Approximate theory formation: an explanation-based approach. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 570–574, St. Paul, MN, 1988. Morgan Kaufmann.

[9] Chi, M., Glaser, J., and Farr, M. *The Nature of Expertise*. Lawrence Elrbaum Associates, Hillsdale, N.J., 1988.

[10] Lenat, D. B., Sutherland, W. R., and Gibbons, J. Heuristic search for new microcircuit structures: an application of artificial intelligence. *AI Magazine*, 3(3):17–33, 1982.

[11] Clancey, W. J. Heuristic classification. *Artificial Intelligence*, 27(3):289–350, 1985.