A Perspective on Formal Verification

Rose F. Gamble
Department of Mathematical and Computer Sciences
The University of Tulsa
Tulsa, OK 74104
gamble@euler.mcs.utulsa.edu

Formal methods employ mathematical techniques to prove a program satisfies its specifications. Many have stated that the use of these methods provides the only means to truly guarantee a program is dependable [4,8]. Though the use of these methods can be susceptible to human error, they will still increase the dependability of a program. Additionally, using formal methods can uncover design problems or inefficiencies in a program. If knowledge base system (KBS) dependability is a goal, formal methods must become as integral a part of system development as software testing. However, we must overcome many obstacles before realizing this goal. The main objective of our research is to address the verification of knowledge based systems using formal methods. For the remainder of this paper, we outline (i) how formal methods can be used to verify KBSs that rely on rule-based programs, (ii) how these methods address issues of efficiency and concurrency, (iii) the lessons learned, and (iv) the future directions of the research.

The use of formal methods. To employ formal methods for developing and verifying KBSs, we need a computational model that provides a notation in which to formally state the specification of the KBS and its associated program. In addition, the computational model needs a proof theory for verification, i.e., to show that the KBS complies with a given specification. Currently we restrict the KBS to using only rules for its knowledge. Swarm [3,10], the computation model used for this research, allows us to develop rule-based programs independent of a particular architecture, to verify dependability using an assertional-style proof logic, and to express concurrent operations. Swarm is based on atomic transactions over a set of tuple-like entities. Its syntax and execution semantics differ only slightly from traditional rule-based systems. In Swarm, the computation state is represented by the contents of a dataspace, a set of content-addressable entities partitioned into a tuple space and a transaction space. Tuples in Swarm are similar to working memory elements. Executing a Swarm transaction is similar to executing a rule in a rule-based program. Tuples and transactions are defined as classes, whose instances can be created dynamically to appear in the tuple and transaction spaces respectively.

A Swarm program executes by non-deterministically choosing a transaction from the transaction space and deleting it from the space. To ensure fairness, every transaction appearing in the transaction space is eventually chosen. There is no notion of global control or sequencing. A transaction may comprise multiple simultaneously executing subtransactions, each of which has a typical rule structure. Given a transaction chosen from the transaction space, the LHSs of all of its subtransactions are simultaneously matched against the tuple space. Those subtransactions whose LHSs successfully match within the current contents of the dataspace perform the actions of their RHSs simultaneously, with all deletions before additions. The combined actions of the executing subtransactions determine the actions of the whole transaction. Tuples can be added and deleted, but transactions can only be added as an action of the transaction itself or as the result of executing another transaction. A Swarm program terminates when there are no transactions left to choose.

We have shown that Swarm programs can be directly translated to traditional rule-based programs. In addition, traditional rule-based programs can be translated to Swarm with the appropriate conversions of the programs' conflict resolution strategies [5]. Swarm is robust in that we can express both synchronous and asynchronous computations, and reason over both the tuple space and transaction space.

Our methodology presupposes the ability to specify the operational details and formal properties of the program under development and to formalize the functional requirements imposed by the application. We specify and verify safety and progress properties using the Swarm proof logic. Safety properties are used to guarantee that the program will do nothing wrong. Progress properties are used to guarantee that the program will eventually do something. Properties are formulated and expressed by constructing a sufficiently complete assertional-style characterization of programs representing possible realizations of an application. For example, we express the verification concern of consistency as a safety property that characterizes the information that cannot be present at the same time in working memory. To prove such a property, we must examine all transactions to show that they do not violate the safety property. We express the verification concern of cycles as one or more progress properties. For instance, one progress property may be that the completion of a certain task eventually leads to the start of another. If we can show that the transactions are such that this property is obeyed, then there are no infinite cycles between the two tasks.

Concurrency and efficiency. The term concurrency describes the possibility of several tasks executing at once. This term is used in contrast to parallelism that refers to the implementation of tasks distributed to several processors. In general, there are at least three reasons to consider concurrency [14]. First, the real world is concurrent. Therefore, if we are modeling real-world tasks, as is often the case in knowledge based systems, concurrency will be inherent in the solution. A system that expresses and exploits the inherent concurrency in a problem solution will realize a performance gain over sequential implementations. Often fault tolerance or robustness requirements may be best met by replication of not only information but also the means of processing. For example, in Swarm, we can partition multiple transaction instances of the same class over a set of variable bindings to replicate knowledge.

Program derivation refers to a systematic formal process of constructing programs from their specifications, typically through some form of stepwise refinement. For concurrent programming, Chandy and Misra's work in UNITY [2] advocates an approach in which the formal program specification is gradually refined to the point where the specification is restrictive enough to suggest a trivial translation into a concurrent program. Thus, program development using this approach first focuses on the problem at hand and postpones language and architectural considerations until last. Back and Sere [1] have a different approach in which they start with an initial (mostly sequential) program and refine its text to represent an efficient concurrent program.

We derive executable rule-based programs from formal specifications by tailoring a methodology that incorporates both the specification refinement and program refinement strategies. Specification refinement is used to derive a generic rule-based program from its formal specifications. Program refinement is used to exploit implementation concerns, such as query complexity and language restrictions, in an effort to maximize efficiency and concurrency. The following are sample refinements from the methodology [7,11]: (i) a general specification refinement, (ii) a general program refinement (iii) a refinement to exploit concurrency, and (iv) a refinement to impose target environment restrictions to increase efficiency.

(i) Often we can divide global progress properties into multiple stages of computation (tasks) or multiple preconditions (cases) of the property, allowing the definition of local progress properties that are easier to express and measure, and hence, easier to guarantee.

(ii) Global data structures used for initial representation often cause bottlenecks. We can distribute these structures locally to reduce shared information, provided we can verify the local computation eventually becomes equivalent to the global computation. This process can introduce new progress properties and force the weakening of safety properties that constrain the structure.

(iii) For more asynchronous computation, we partition a transaction modifying several variables into a set of transactions, each modifying a few variables. We must verify that the distributed modification of variables eventually becomes equivalent to the global modification.

(iv) With knowledge of the target system we can impose restrictions early in the refinement to still derive an efficient concurrent program. For instance, if the target system is a parallel production system shell such as PARS we need consider the difference in the atomicity of the knowledge statement. In Swarm a transaction is an atomic statement. However, conversion to PARS requires a subtransaction to be an atomic statement.

Lessons learned. We approached verification from two viewpoints: verifying existing systems and deriving verifiable systems from specifications. The former requires some reverse engineering because many specifications needed for proof may not be explicitly stated. Also, verifying an entire KBS is a complex and time-consuming process. However, if economy of resources is important, it is possible to verify only portions of a large program [6], allowing concentration of formal methods where it is critical that code be dependable. Deriving a KBS is a more direct means of verification because the proof of the system is an inherent consideration of each refinement step. Knowledge of the target environment can offer substantial information that influences the derivation process. Specification refinement allows us to view the program generically, while program refinement offers a way to incorporate the constraints and exploit the advantageous characteristics of the target environment.

We found that we can best exploit concurrency in rule-based programs through derivation. This method contrasts with methods used by parallel production system shell developers [9,13] that partition rules of existing sequential programs for multi-rule execution. Concentration on the sequential program inhibits its concurrency because it restricts the choice of solution strategies that may increase the number of co-executing rules.

Future directions of research. Formal methods have been criticized for the amount of resources they require. These resources are normally in the form of manual specification, refinement, and proof. For formal methods to be commercially accepted for KBS development, we envision two immediate directions for research. One direction is to abstract the proof methodology and refinement process and to create a partially automated development tool. The second direction is to look at current methods for specifying KBSs [5,12,15] to determine if those specifications that are crucial to the acceptance of the KBS in a commercial setting can be readily found, and then to concentrate formal methods only on the portions of the program responsible for satisfying the crucial specifications. The program as a whole can be rigorously tested to satisfy the remaining specifications.

Knowledge base systems made up of purely rule-based programs are not robust enough. Many vendors offer software development environments that include both objects and rules. We can model concrete classes and instances within Swarm. Additionally, an object can be modeled as an independent process with transactions as methods, to maintain autonomy and to concurrently execute with other objects. However, we need to extend Swarm to combine multiple objects into one system and to use inheritance. Currently, we are enhancing our methodology to derive KBSs to use objects and rules. Through this research we hope to determine how to subject this integrated paradigm to formal verification and derivation.

References

[1] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13:133-180, 1990.

[2] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, New York, 1988.

[3] H.C. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. *IEEE Transactions on Parallel and Distributed Systems*, 1(3): 365-376, 1990.

[4] E.D. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[5] R.F. Gamble, G.-C. Roman, W.E. Ball, and H.C. Cunningham. The application of formal specification and verification to rule-based programs. *International Journal of Expert Systems Research and Applications*, 1992, (to appear).

[6] R.F. Gamble, G.-C. Roman, and W.E. Ball. Formal verification of rule-based programs. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pp. 329-334, July 1991.

[7] R.F. Gamble and G.-C. Roman. Achieving software reliability in concurrent expert systems. Manuscript in preparation.

[8] A. Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice-Hall, New York, 1990.

[9] C.M. Kuo, D.P. Miranker, and J.C. Browne On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13(4): 424-441, 1991.

[10] G.-C. Roman and H.C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361-1373, December 1990.

[11] G.-C. Roman, R.F. Gamble, and W.E. Ball. Formal derivation of rule-based programs. *IEEE Transactions on Software Engineering,* 1992, (to appear).

[12] J. Rushby and R.A. Whitehurst. Formal verification of AI software. Technical report, SRI International, Computer Science Laboratory. February 1989.

[13] J.G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13(4):348-365, 1991.

[14] C. Tomlinson and M. Scheevel. Concurrent Object-Oriented Programming Languages. In W. Kim and F.H. Lochovsky, eds. *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, Reading, Mass. 1989.

[15] J. Yen and J. Lee. A task-based methodology for specifying expert systems. *IEEE Expert*, 8(1): 8-15, 1993.