Swarming over the Software Barrier

H. Conrad Cunningham
Department of Computer and Information Science
University of Mississippi
University, MS 38677 U.S.A.

Abstract

Swarm is a concurrent programming model which integrates a Linda-like communication medium, the shared dataspace*, with a UNITY-like computational model, proof system, and program structure. It generalizes the Linda tuple-space operations by providing more powerful dataspace queries. It generalizes UNITY by permitting content-based access to data, a dynamic set of statements, and the capability to control the execution mode (i.e., synchronous or asynchronous) for arbitrary collections of program statements.*

1 Introduction

Although the parallel machines and networks that have been developed in recent years offer the potential for enormous increases in computational power, our capabilities for harnessing this power is still quite primitive. We are bumping up against what Peter Denning has termed the "software barrier" [6]. This software barrier is made more formidable by the fact that each parallel machine typically has its own idiosyncratic programming languages and features. David Gelernter (quoted in [7]) voices the complaint that "programmers have been forced to accommodate themselves to the machines rather than vice versa." He suggests that researchers should imagine new kinds of programs before they imagine new kinds of machines.

With the goal of providing a practical, machine-independent vehicle for programming parallel machines, Gelernter and his colleagues have put forth the Linda communication model [3]. A salient (and appealing) feature of Linda is it's "tuple space"—a common, content-addressable data structure. Concurrently executing processes communicate by inserting tuples into, deleting tuples from, and examining tuples in this tuple space. The tuple space also encompasses the processes themselves; they are viewed as "live tuples" which are inserted into the tuple space by their parent processes. Tuple space communication thus enables a flexible, uncoupled relationship among the data and processing components of the program.

Although their approach is quite different, Mani Chandy and Jay Misra also seek to enable programmers who are mired in the details of programming parallel machines to lift themselves up over the software barrier. They argue that the fragmentation of programming approaches along the lines of architectural structure, application area, and programming language features obscures the basic unity of the programming task [4]. With the UNITY model, they seek to unify seemingly disparate areas of programming with a simple theory consisting of a model of computation and an associated proof system. The UNITY approach separates the concern for functional correctness of the program from the concern for efficient execution of the program on a specific machine. The programmer first develops a correct program from the specifications, then transforms it to execute efficiently on the chosen hardware.

We find aspects of both Linda and UNITY appealing. We like the flexible program/data structures made possible by the Linda tuple space, but also want the benefit of a proof system and program derivation techniques. We like UNITY's simple computational model and proof system, but also want to be able to handle more flexible and dynamic program and data structures. Thus we sought to integrate the "desirable" features of Linda and UNITY into one model. We call the resulting model *Swarm*, evoking the image of a large, rapidly moving aggregation of small, independent agents cooperating to perform a task.

2 Swarm

Swarm [5, 13] is a concurrent programming model which combines a Linda-like communication medium, the *shared dataspace*, with a UNITY-like computational model, proof system, and program structure. It generalizes the Linda tuple-space operations by providing more powerful dataspace queries. It generalizes

UNITY by permitting content-based access to data, a dynamic set of statements, and the capability to control the execution mode (i.e., synchronous or asynchronous) for arbitrary collections of program statements.

The statements in the Swarm notation are called *transactions*. A transaction denotes an atomic transformation of the dataspace. It is a set of concurrently executed query-action pairs. Each query-action pair is similar to a production rule in a language like OPS5 [2]. A query consists of a predicate over the dataspace; an action specifies a group of deletions and insertions of dataspace elements that are done when the query is executed successfully. Instances of transactions are created dynamically by an executing program. They are represented in the dataspace by tuple-like entities.

The execution of a Swarm program is similar to execution of a UNITY program. It begins execution from a specified initial dataspace. On each execution step a transaction is chosen nondeterministically from the dataspace and executed atomically. This selection is fair in the sense that every transaction in the dataspace at any point in the computation will eventually be chosen. An executing transaction examines the dataspace and then, depending upon the results of the examination, can delete tuples (but not transactions) from the dataspace and insert new tuples and transactions into the dataspace. Unless a transaction explicitly reinserts itself into the dataspace, it is deleted as a by-product of its own execution. Program execution continues until there are no transactions remaining in the dataspace.

As a simple example, consider the program in Figure 1 for computing prime numbers using the sieve algorithm. The program uses tuples of type *num* to hold the candidate prime numbers and transactions of type *Sieve* to filter nonprimes from the set of candidates. The program begins execution with a dataspace containing a *num* tuple and *Sieve* transaction for each integer in the range 2 through N. If an executing transaction finds a *num* tuple holding a multiple of the transaction's parameter value, then it deletes the tuple (denoted by the † operation) and reinserts itself to search for more multiples; otherwise, the only action taken is the deletion of the transaction itself from the dataspace. Thus, when there are no more nonprimes to delete, the program halts.

Like UNITY, the Swarm proof system uses an assertional programming logic which relies upon proof of program-wide properties, e.g., global invariants and progress properties [5]. We define the Swarm logic in terms of the same logical relations as UNITY (**unless**, **stable**, **invariant**, **ensures**, and **leads-to**), but must reformulate several of the concepts to accommodate Swarm's distinctive features. We define a proof rule for transaction statements to replace UNITY's well-known rule for multiple-assignment statements, redefine UNITY's **ensures** relation to handle the creation and deletion of transaction statements, and replace UNITY's use of fixed-point predicates with other methods for determining program termination. We have constructed our logic carefully so that most of the theorems developed for UNITY can be directly adapted to the Swarm logic.

```
program $Prime\,(N : 2 \leq N)$
    tuple types
        $[\, i : 2 \leq i \leq N :: num(i)\, ]$
    transaction types
        $[\, i : 2 \leq i \leq N ::$
            $Sieve(i) \equiv$
                $j : num(j)\dagger,\ i < j,\ j \ \mathbf{mod}\ i = 0$
                    $\longrightarrow Sieve(i)$
        $]$
    initialization
        $[\, i : 2 \leq i \leq N :: num(i),\ Sieve(i)\, ]$
end
```

Figure 1: Computing Prime Numbers

A specification for the prime number program in Figure 1 can be given in terms of two properties: a progress property stated using the relation **leads-to** and a safety property stated using the relation **stable**. Let *Post* represent the desired postcondition for the program, the predicate

$$\langle \forall\, i \ :: \ num(i) \ \equiv \ 2 \leq i \leq N \wedge prime(i) \ \rangle$$

where $prime(i)$ denotes *true* when i is a prime number and *false* otherwise. (*Post* means that the tuple $num(i)$ exists if and only if i is a prime number in the range 2 through N.) From any point during its execution, the program must eventually make *Post* true. In the Swarm programming logic this can be stated as

$$\mathit{true} \ \ \mathbf{leads\text{-}to} \ \ \mathit{Post}.$$

Once achieved, the program must remain in a state satisfying this postcondition, i.e.,

$$\mathbf{stable} \ \ \mathit{Post}.$$

Although the set of transactions in the dataspace (usually) changes as a program executes, the set of query-action pairs within a transaction is static. The notion of dynamically constructing "transactions" from arbitrary sets of query-action pairs seemed to be an appealing extension of the Swarm concept. We incorporated this capability into Swarm by means of the *synchrony relation* feature [13, 14]. The synchrony relation is a symmetric relation on the set of possible transactions in the program. This relation may be examined and modified by the program in the same way that other parts of the dataspace (i.e., data tuples and transactions) can be. To accommodate the synchrony relation, we extend the program execution model in the following way: whenever a transaction is chosen for execution, all transactions in the dataspace which are related to the chosen transaction by (the closure of) the synchrony relation are also chosen; all of the transactions that make up this set, called a *synchronic group*, are executed as if they comprised a single transaction. We have generalized the programming logic to handle synchronic groups as the atomic statements [14].

By enabling asynchronous program fragments to be coalesced dynamically into synchronous subcomputations (i.e., dynamic partial synchrony [16]), the synchrony relation provides an elegant mechanism for structuring concurrent computations. This unique feature facilitates a programming style in which the granularity of the computation can be changed dynamically to accommodate structural variations in the input. This feature also suggests mechanisms for the programming of a mixed-modality parallel computer, i.e., a computer which can simultaneously execute asynchronous and synchronous computations. Perhaps architectures of this type could enable both higher performance and greater flexibility in algorithm design.

3 Discussion

Swarm emerged in its current form in late 1988 as a part of this author's collaboration with his doctoral advisor, Catalin Roman. Previously we had been experimenting with the Shared Dataspace Language (SDL) [15], a CSP-like [9] language in which processes communicate through a Linda-like [3] tuple space. Becoming dissatisfied with the complexity of SDL, we sought a simpler notation and a better formal foundation for our shared dataspace research. We became intrigued with the UNITY [4] programming model and decided to adopt its approach for our new research vehicle. By merging the shared dataspace concepts from SDL into a UNITY-like computational model, we constructed a new model and notation that was somewhat reminiscent of production rule languages like OPS5 [2]. Because of the dynamic structure of the programs, we decided to call the new model and notation Swarm.

Many of the elements of Swarm have appeared in other languages designed for parallel execution. Like Swarm programs, programs in the Associons [11] model manipulate sets of tuples. In this notation programs consist of a sequence of deterministic "closure" statements which add new tuples to the set while their guards are true. The recent work on the GAMMA model [1] is similar in spirit to Associons. However, GAMMA differs in that programs manipulate bags of entities with a nondeterministic "Γ" operator. The GAMMA model also provides a program derivation methodology. In both Associons and GAMMA the parallelism is implicit—hidden in the powerful closure and Γ operators. On the other hand, Swarm's synchrony relation allows a program to modify the structure of its parallelism explicitly to conform to the structure of its data or other aspects of its operating environment.

Swarm has proven to be an excellent research vehicle. We have defined the Swarm programming notation and specified a formal operational model based on a state-transition approach [13]. To facilitate formal verification of Swarm programs, we have developed an assertional programming logic and devised proof techniques appropriate for the dynamic structure of Swarm [5, 14]. The Swarm ideas have motivated an exploration of new approaches to algorithm development and programming methodology [13, 16] and to program refinement [10].

Swarm continues to stimulate new ideas. In a related effort, colleagues are investigating the use of the shared dataspace model as a basis for a "declarative" approach to the visualization of the dynamics of concurrent program execution [12]. The Swarm notation and logic are also being applied to the design and verification of production rule systems [8]. We also believe Swarm has considerable potential as a conceptual framework for software engineering. To realize this potential, however, the Swarm theory and methods need to be extended to aid systematic development of large programs. With this in mind, we have recently begun to study program composition and refinement, derivation of programs from specifications, and transformations of programs toward various processor architectures.

4 Conclusion

This paper began with a discussion of a "software barrier"—an intellectual and technological obstacle obstructing our path toward the full exploitation of the power of parallel computers. Swarm, in its current form, is not a magic bulldozer which will push aside the barrier. However, we do believe that Swarm has been, and still is, a good reconnaissance vehicle. Perhaps, by exploring the barrier further, we will find that there are convenient paths over, around, or through the barrier—or maybe our intensive study of the barrier will reveal that it is merely a figment of our (lack of) imagination.

Acknowledgements

We thank the two anonymous referees and Catalin Roman of Washington University in St. Louis for their helpful suggestions concerning this paper.

References

[1] J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[2] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts, 1985.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[5] 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, July 1990.

[6] P. J. Denning. Editorial: Parallel computing and its evolution. *Communications of the ACM*, 29(12):1163–1167, December 1986.

[7] K. A. Frenkel. Introduction: Special issue on parallelism. *Communications of the ACM*, 29(12):1168–1169, December 1986.

[8] R. F. Gamble, G.-C. Roman, and W. E. Ball. Formal verification of pure production system programs. In *Proceedings of the Ninth National Conference on Artificial Intelligence* (AAAI-91), pages 329–334, July 1991.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[10] Y. Liu and A. K. Singh. Parallel programming: Achieving portability through abstraction. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, May 1991.

[11] M. Rem. Associons: A program notation with tuples instead of variables. *ACM Transactions on Programming Languages and Systems*, 3(3):251–262, July 1981.

[12] G.-C. Roman and K. C. Cox. A declarative approach to visualizing concurrent computations. *Computer*, 22(10):25–36, October 1989.

[13] G.-C. Roman and H. C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–73, December 1990.

[14] G.-C. Roman and H. C. Cunningham. The synchronic group: A concurrent programming concept and its proof logic. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 142–149. IEEE, May 1990.

[15] G.-C. Roman, H. C. Cunningham, and M. E. Ehlers. A shared dataspace language supporting large-scale concurrency. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 265–272. IEEE, June 1988.

[16] G.-C. Roman, J. Y. Plun, and C. D. Wilcox. Dynamic partial synchrony. Technical Report 90–36, Washington University, Department of Computer Science, St. Louis, Missouri, October 1990.