Goal-Oriented Programming, or Composition Using Events, or Threads Considered Harmful

Robbert van Renesse <rvr@cs.cornell.edu> *
Department of Computer Science
Cornell University, Ithaca, NY 14853

Many applications, and particularly distributed applications, are about dealing with events. The code is of the form: when this message arrives do this; when this timer expires do that; when that server fails do something else; etc. Perhaps the most natural way to program this is by using interrupt handlers. Programmers have found this hard to program, and prefer loops of the form: `for ever { e = await_event(); handle_event(e); }`. But there are several problems with this approach. The `handle_event` routine may take a long time to execute, slowing down the handling of subsequent events. Worse, `handle_event` may invoke `await_event` to wait, for example, for the reply of a Remote Procedure Call.

To deal with this, the *thread* abstraction was introduced. While threads are handling events, or awaiting specific events, unrelated events can be handled by other threads. Unfortunately, anybody who has ever used threads extensively will agree that threads are error prone, hard to debug, and often non-portable. Composing packages that use different thread abstractions is usually not possible or extremely hard. The only widely popular language that supports threads well, Java, is still rarely used for serious distributed systems applications. Even there, simple applets that use threads often have bugs in them, in that they do not stop when they are off the screen, do not start up correctly when they reappear, or have race or deadlock conditions when they access shared object instances.

I believe that `thread_fork` is the `goto` equivalent of parallel programming. Both request a transfer to a particular entry in your program, with no indication of scope. Once you use a bunch of these in your program, any underlying logical structure is now lost, and the program becomes hard to understand and debug. Another basic problem is that available thread synchronization paradigms are either too low-level, or too coarse-grained. Low-level mechanisms like semaphores are very error-prone. Coarse-grained mechanisms like monitors do not allow for much parallelism by overspecifying the amount of synchronization required.

An alternative idea is to go back to programming directly with events. Although many event management and notification systems are under development that generate, filter, and deliver events (among many examples, Corba [5] and Yeast [4]), they do not provide much help with writing a program that takes events as input. Below, I will propose a new programming technique that I call *goal-oriented programming*, which goes back to the roots of step-wise refinement of programs. I will show that this technique is not only easier than programming with threads, but it also provides better performance and portability. I will illustrate this by means of an example that I will work out first using threads, then using goal-oriented programming. I will conclude with a description of my experience with a transactional file system I have built this way.

Example

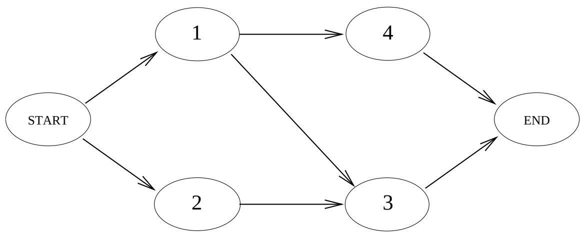
The example involves a simple directory service that maps names to object identifiers. In our example, we create an object (returning an object identifier), and then add a (name, identifier)-mapping to the directory service, as well as applying a relatively expensive operation on the new object. The directory server, object server, and application each run on a different machine. The application uses Remote Procedure Call to communicate with the servers.

There are four primitive actions that need be performed:

*This work was supported by ARPA/ONR grant N00014-92-J-1866

1. creating the object, which returns the object identifier
2. reading the current directory
3. updating and writing back the directory
4. doing some operation on the object

There are dependencies between these operations (that is, some have to complete before others can start), which we can depict graphically as follows:



As can be seen from this graph, some parallelism of operations is allowed, but the description of this is somewhat complex because the parallelism is not well-nested. In particular, you cannot express all parallelism using `cobegin-coend`.

In general, we will not know how expensive the operations are. This may be different from time to time. For example, the directory may or may not be cached. Let us initially assume that creating an object is cheap, but that the directory is not cached and therefore requires an expensive disk operation. The updated directory is written back to the cache and is therefore cheap. Finally, we assume that the operation on the object is about as expensive as reading the directory. These assumptions will influence the way we implement the application.

Implementation Styles

Before threads (and still the most common practice), a programmer would serialize the four operations consistent with the partial order implied by the given dependencies. E.g., { `op1; op2; op3; op4;` } and { `op2; op1; op4; op3;` } will both work, while { `op2; op3; op1; op4;` } would be incorrect.

With threads, we may first try to exploit parallelism completely. Thus we would immediately fork to create threads for operations 1 and 2 to run in parallel. When operation 1 completes, it can increment a semaphore (initialized to 0) for operation 3, and start operation 4. When operation 2 completes, it decrements the semaphore for operation 3 (therefore blocks if operation 1 hasn't completed yet), and then executes operation 3. The same thing happens after operations 3 and 4. For example, the first thread may try to decrement the semaphore before completing, while the second increments it, and then exits.

Although not impossible, this requires quite a bit of sophistication on behalf of the programmer. He or she has to realize that two semaphores are needed for synchronization, and figure out which threads should acquire or release the semaphores at which points in time. Here there are only four operations, but (as we shall see later), in a more realistic program there will be many such synchronization points and it becomes very unclear how many threads and semaphores are required.

More likely, the programmer will think to run the first two operations in parallel, await their completion, and then run the next two, and again await their completion. This way he or she does not have to anticipate the second semaphore until the first operations have completed. Unfortunately, if our assumptions are correct, this approach does not help performance much, since the long running operations 2 and 4 would not run in parallel. A better solution, in this particular case, is to run first operation 1, then operations 2 and 4 in parallel, and finally operation 3. But this is unintuitive for several reasons, since we do not appear to use the threads to their fullest (only two operations are executed in parallel once, rather than twice). Also, if our assumptions are wrong, for example because the directory is cached, this strategy would *not* be optimal. In general, any such simplification overspecifies the synchronization points and can lead to unoptimal performance.

Goal-Oriented Programming

My suggestion is to return to programming principles by designing and implementing programs *top-down*, using stepwise refinement, rather than serially (*do this, then that, etc.*). However, rather than accomplishing this using procedures, I am suggesting to define a series of goals and the dependencies between the goals. I call this *goal-oriented programming*. The basic idea behind goal-oriented programming is the same as that of data-flow engines, except applied at the level of programming. The programming language that comes closests to supporting this is perhaps Prolog [3]. As I will demonstrate, goal-oriented programming can be easily adopted in other languages, such as Java. Goals can include reading a directory or writing a file. The completion of a goal constitutes an event.

For each goal, the programmer specifies the subgoals that are necessary to accomplish first, and the action that is necessary to accomplish the goal. To the best of my knowledge, the best known way to describe goals, their dependencies, and their actions, is using a *Makefile* format. For our example, the program would look like this:

```
Final-Goal: Goal-3 Goal-4
        Done

Goal-4:     Goal-1
        Some expensive operation on object

Goal-3:     Goal-1 Goal-2
        Update directory; Write back

Goal-2:
        Read directory

Goal-1:
        Create object
```

Each rule specifies dependencies and actions. Actions are written in a standard programming language (I currently support Java and ML). Each action is asynchronous, and generates an event when completed (in all these cases, this would be the reply to an RPC). The value of the event would be bound to the goal. For example, Goal-3 depends on the results of Goal-1 and Goal-2. When both have completed, Goal-1 is bound to the object identifier, and Goal-2 to the contents of the directory. The action of Goal-3 then calculates the contents of the new directory, and writes back the result. When done, it generates the Goal-3 event that Final-Goal requires. (Preferably, the names of the goals would have descriptive names.)

All synchronization is done automatically without the use of threads or semaphores, much like in implementations of parallel make [1] and the inference engine of a parallel Prolog system (for example, [6]). The program is therefore highly portable and efficient, and I believe that both parallelism and synchronization are specified at the right level of abstraction.

Performance

Figure 1 compares the performance of these three styles of implementation of this example. The first style is running the primitives sequentially. The second style uses a very efficient home-brew thread implementation, and runs operations 1 and 2 in parallel, followed by operations 3 and 4 in parallel. The last style is goal-oriented programming. The examples were run over the U-Net ATM interface, which provides one-way latencies of about 35 microseconds [2]. On the x axes is the time that step 2 and 4 take in microseconds. The delays were generated artificially. Operations 1 and 3 require a short round-trip over U-Net, but are otherwise of almost null cost. On the y axes is the time to completion in microseconds. Both scales are logarithmic.

Unsurprisingly, the goal-oriented program performs best, and the sequential program performs worst. Perhaps more interestingly, the threaded program performs as well as the goal-oriented one if all operations are very short (less than 50 microseconds). This contradicts common folklore that threads are not well-suited for fine-grained parallelism.

Language Extension or Library?

As mentioned, I currently support goal-oriented programming both in ML and in Java. For ML, I have written a preprocessor that transforms a goal-oriented program like this into a state machine expressed in ML (much

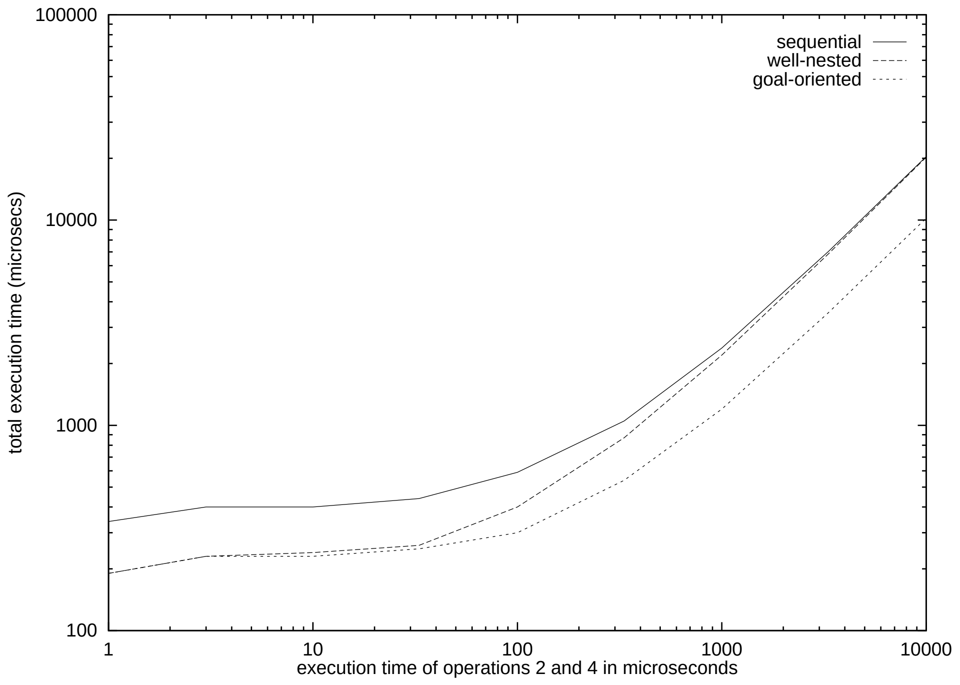


Figure 1: The total completion time of our example as a function of the length of operations 2 and 4, for each programming style in which the example was implemented.

like a compiler-generator such as YACC). For Java, I have not created a variant of the language, but instead added library support. In doing so, I discovered advantages and disadvantages of these approaches.

The preprocessed variety can generate more efficient code, because the results of dependencies can be accessed directly through a parameter to the body of the rule, rather than through a indirect call to a library routine. Also, the program looks much cleaner, partially because of the direct access to dependent results, and also because no initialization is necessary. On the other hand, library calls result in much more verbose programs, but support dynamic generation of rules and dependencies, resulting in better flexibility. For example, it allows the generation of a rule to read a particular set of files. This particular case *is* supported in the preprocessed variety through a library routine that can activate a set of rules and await all results, but it is clear that dynamic generation of rules provides more power to the programmer. (The Java library support contains only 160 lines of code.)

Exceptions and Nesting

Actions are allowed to raise *exceptions*. The value associated with the exception is used as the goal's value. Other rules that depend on this goal can test to see if a goal resulted in an exception. If they neglect to do so and access the goal's value otherwise, they themselves will throw an exception.

As in object-oriented programming, where you can have several instances of the same class, you can have several instances of the same goal active. This is accomplished in much the same way as the common practice of invoking *make* from a Makefile. The result event of such a nested make is then used as the result of the rule that invoked it. *Nesting* is particularly useful for conditional evaluation of goals and tail-recursive loops.

As an example of conditional evaluation, say that we want to read a file, but that we keep a local cache. Obviously, we don't want to retrieve the file from the server if it's in the cache. The way that can be specified is as follows:

```
ReadGoal: FileName
              try {
                   ReadFromCache(Filename);
              } catch (NotFound) {
                   make ReadFromFileServer;
              }

ReadFromFileServer: FileName FileServer
              rpc(FileServer, "READ", FileName);
```

Tail-recursion is useful when, for example, we wish to support path names. In case of reading a file specified by a path name, we first need to retrieve the top-level directory, then the sub-directory, etc., and finally the file itself. This can be specified as follows:

```
ReadPath: ReadGoal PathName
              (dir, rest) := split(PathName);
              if (rest == "")
                   ReadGoal;
              else
                   FileName=dir PathName=rest make ReadPath;
```

The last statement specifies that `ReadPath` should be evaluated again, but this time with `FileName` bound to the first component of the path, and `PathName` bound to the rest of the path.

Both the ML and Java extensions actually support two different forms of nested make. In the first version, the nested make is evaluated in the same environment, with already evaluated goals intact. In the second, a new environment is created, in which all goals start out uninitialized.

Experience

I have written the user interface to a home-brew transactional file system using goal-oriented programming. The program contains 77 goals. It uses nesting in six places for achieving conditional evaluation and loops. The dependencies are easy to specify, but the resulting dependency graph is very detailed and allows for much parallelism. Specifying all parallelism with threads would have been undoable, and figuring out how to reduce it to only a few threads non-obvious. On the other hand, the preprocessor extracts all parallelism automatically. My experience is that the technique is powerful, but needs more work and thought. Here are some ideas that I wish to explore.

First, a *goal browser*, much like an object-oriented *class browser*, would simplify the discovery and re-use of existing goals. Second, there is no reason why all the goals and dependencies are only specified on a per-process basis: the entire distributed application can be described this way, and dependencies can span process boundaries. This could also obviate the need for RPC. Finally, I believe a debugger for this system can be powerful and easily designed and written, for example allowing breakpoints at particular goals, and logging the order in which operations were started and completed.

Acknowledgments

I would like to thank the referees for their insights.

References

[1] Erik H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, Spring 1988.

[2] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, December 1995.

[3] W. F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, third revised and extended edition, 1987.

[4] B. Krishnamurthy and D. S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10), October 1995.

[5] Open Management Group. Event-condition-action rules management facility. *OMG Document cf/97-01-10*, July 1997.

[6] Victor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I, A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Proc. of the 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.