Modularity and Conventions for Maintainable Concurrent Language Implementations

A Review of Our Experiences and Practices

Stefan Marr Jens Nicolay Tom Van Cutsem Theo D'Hondt

Software Languages Lab, Vrije Universiteit Brussel, Belgium

{stefan.marr, jens.nicolay, tvcutsem, tjdhondt}@vub.ac.be

Abstract

In this paper, we review what we have learned from implementing languages for parallel and concurrent programming, and investigate the role of modularity. To identify the approaches used to facilitate correctness and maintainability, we ask the following questions: What guides modularization? Are informal approaches used to facilitate correctness? Are concurrency concerns modularized? And, where is language support lacking most?

Our subjects are AmbientTalk, SLIP, and the RoarVM. All three evolved over the years, enabling us to look back at specific experiments to understand the impact of concurrency on modularity.

We conclude from our review that concurrency concerns are one of the strongest drivers for the definition of module boundaries. It helps when languages offer sophisticated modularization constructs. However, with respect to concurrency, other language features like single-assignment are of greater importance. Furthermore, tooling that enables re-modularization taking concurrency invariants into account would be of great value.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters; D.1.3 [*Programming Techniques*]: Concurrent Programing; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Reliability

Keywords Modularization, Case Study, Experience Report, Concurrency, Virtual Machines

1. Context and Contributions

This paper reviews our experiences with the implementation of interpreter-based concurrent and parallel languages to assess the role of modularity and its relation to concurrency aspects. We developed or maintained AmbientTalk, SLIP, and the RoarVM during the last decade and did a number of experiments related to this topic during that time. These language implementations serve to validate our research and constitute a foundation for our teaching activities. This specific setting had an important impact on the implementation of these systems. To facilitate experiments and teaching, their core must remain maintainable.

Another aspect of this setting is the focus on interpreters instead of classic compilers or performance oriented just-in-time compiling approaches. The most important benefit of the former is their malleability. Therefore, interpreters are accessible for students and require relatively low engineering effort for experiments. On the downside, their performance usually remains suboptimal. However, for the sake of discussing modularity, these language implementations have sufficient complexity.

The goal of this paper is to review our approaches to modularity when it comes to issues of concurrency and parallelism. To that end, we analyze how we use language mechanisms and conventions to tackle problems like locking, shared state, safe messaging, and global synchronization. First, we detail the questions we try to answer for each subject of our case study. This is followed by an overview of the subjects, i. e., our language implementations. Finally, we discuss the results and our conclusions.

2. Case Study Design

For the review of our language implementations, we will do a general review of the applied practices, as well as a review of specific experiments which will give us a more detailed insight into the modularity of changes.

General Review We focus in our reviews on informal implementation aspects like coding conventions and the impact of parallelism and concurrency on modularization decisions. To guide the reviews we use the following questions:

- Are modularization decisions guided by a specific rule or principle that takes concurrency aspects into account?
- Are informal approaches used to handle concurrency aspects like data-races, deadlocks, or performance issues?
- Do concurrency aspects span multiple subsystems or are they localized to specific ones?
- Are the available modularization constructs sufficient to suitably and consistently handle concurrency aspects?

Experiment Review For more insights into how concurrency aspects and parallel execution influence the modularization, we review the implementation impact of a number of experiments. In addition to the general ones, these reviews are guided by the following questions:

- Do the changes exhibit cross-cutting characteristics, i. e., how modular is the introduced functionality?
- Have key subsystems been remodularized to accommodate for new functionality and its invariants?
- Could advanced mechanisms for modularity have helped to achieve better maintainability especially with regard to concurrency-related invariants?

3. Case Study Subjects

AmbientTalk [4] is a distributed object-oriented language that can be regarded as a scripting language for mobile phones, designed to build applications for mobile networks. AmbientTalk uses actor-based message-passing concurrency and distribution. The AmbientTalk VM is written in Java and targets Android-based smartphones.

One aspect of the AmbientTalk VM is that it internally uses *event loop* concurrency. Event loops are similar to actors but do allow shared mutable state in general. They are used for instance for VM-level services that need to run in their own thread of control, such as service discovery or network communication services. The main means of communication is similar to that of actors based on message passing.

RoarVM Ungar and Adams [9] built the RoarVM, a Smalltalk for the Tilera TILE64 manycore system, providing application-level parallelism on a shared-memory heap. To achieve acceptable performance, the heap is divided into read/write and read-mostly memory, which enables more efficient use of the hardware cache coherency. Experimentation with the non-uniform memory access properties is facilitated with primitives for explicit object movement between cores. It is used to explore a new style of non-deterministic programming that harnesses emerging behavior [8].

In this paper, we examine the following experiments:

Threads vs. Processes Porting the RoarVM to classic multicore systems led us to use threads instead of processes. The main reasons are the missing library support and the absence of sufficient debugging tools for process-based applications.

Work-Stealing Scheduling The RoarVM uses a single scheduler for compatibility with standard Smalltalks. On manycore architectures, this design becomes a bottleneck and was replaced by a work-stealing scheduler with a more scalable design [1].

SLIP is a platform for the instruction and exploration of language implementations. The main focus is not on the language itself, which is a minimal but complete version of LISP with a distinct Scheme flavor. Instead, SLIP is a sequence of interpreters, starting with a 100 line fully metacircular version. This version is rewritten in continuation passing style and translated into C as a straightforward AST interpreter. Fourteen successive versions introduce features such as trampolines, lexical addressing, and garbage collection, ending up with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

Multi-Threading Support Version 14 introduces threads. We discuss here the approach to modularity and its implication on the overall architecture of the implementation.

4. Case Study Reviews

In this section, we discuss the results for each language implementation.

4.1 AmbientTalk

The guiding principle to structure concurrency in the AmbientTalk VM is the same as the one used in the AmbientTalk language itself: avoid shared mutable state and decouple modules by relying on asynchronous communication.

4.1.1 Event Loop Concurrency Design

Traditionally, multi-threaded applications are partitioned into multiple *threads*, which execute an arbitrary piece of code concurrently and which synchronize and communicate implicitly by means of *shared data*. Multi-threading is hard to program and debug because threads may interact non-deterministically, requiring the programmer to think about all possible operation interleavings on the shared data.

Concurrency based on event loops operates by a different principle. Here, each thread executes what is known as an *event loop*, a perpetual loop that takes incoming events from an *event queue* and processes them one by one. Event loops communicate using *asynchronous message passing*: one event loop puts a message in the event queue of the recipient, which will process it at a later point in time.

In itself, event loops do not protect a program from race conditions: multiple event loops may still operate on the same piece of mutable data. To banish data races, each piece of data is assigned an *owning* event loop, and only the owner may operate on its data synchronously. Other event loops may only modify the data indirectly (and asynchronously) by asking the owner to perform the modification instead.

Concurrency in the AmbientTalk VM is built around an event loop concurrency framework. There are different kinds of event loops in the AmbientTalk VM: some represent AmbientTalk actors, others represent VM-level services, mostly to process incoming and outgoing network traffic.

4.1.2 Implementation

Event loop concurrency requires that the entire codebase follows a strict set of rules in order for its properties to hold. Since these rules have a crosscutting impact, we will go into more detail on how they affect the code.

All event loops inherit from a common superclass to both classify instances of particular classes as event loops, and to reuse their implementation (including thread and message queue handling, and default message processing behavior).

To further identify event loops, we require that all classes inheriting from the common base class are prefixed with `EL`. This naming scheme, while merely a convention, makes it clearer in the code when one is dealing with active objects.

Methods in event loop classes whose invocation triggers an event that is sent to an event loop, are prefixed with `event_` for asynchronous and `sync_event_` for synchronous event notifications. Again, it is merely a naming convention.

Since the parameters passed to an `event_` or `sync_event_` method become shared by both the sender and receiver event loop, care must be taken that both event loops properly synchronize access to this shared state. In staying true to the "no shared state" philosophy of the actor model, we require that these parameters are read-only for the sender. That is: either the passed data is immutable, or it is mutable, but the sender promises to no longer update the data once sent, granting the receiver exclusive ownership of the data. In Java, there is no language mechanism that could enforce this constraint. In this case, the programmer must manually inspect all relevant method signatures and see if the parameter-passed types adhere to this restriction.

Since every event loop runs in its own Java thread, mutable state that is "global" within a single event loop is implemented with thread-local variables. Every event loop operates on its own set of values. From a modularity point of view, the use of thread-local variables avoids parameterizing all event loop dependent code with their values.

4.1.3 Conclusion

Given the nature of the AmbientTalk VM as a runtime for a concurrent programming language, concurrency is not localized to a specific subsystem. On the other hand, by representing all concurrent activities as event loops and virtually all concurrent communication via explicit (asynchronous) message passing, concurrency becomes manageable even though it is pervasive throughout the implementation. Since Java has no language features to identify event loops or asynchronous message passing, we resorted to naming conventions instead. The biggest challenge was to partition the shared-memory Java heap such that event loops always have exclusive write access to all data they have access to. We had to resort to manual inspection of the types of all relevant method signatures to enforce this invariant. More elaborate type systems such as ownership types [3] could alleviate this concern.

4.2 RoarVM

In the RoarVM implementation, Ungar and Adams applied principles very similar to the event loop model used in the AmbientTalk VM. Every *interpreter instance*, i. e., every bytecode interpretation loop, runs in its own operating system process, making memory private to the interpreter instance by default. Communication between interpreter instances is done with a mix of explicitly requested shared memory and message passing.

4.2.1 On-demand Shared Memory and Asynchronous Message Passing

The main entities, which also correspond to implementation modules, are the interpreter instances. Beside executing the classic bytecode interpreter for Smalltalk, they also act as event loops to enable communication with other interpreter instances when it comes to ensuring global invariants pertaining to garbage collection or scheduling policies.

Modules can adapt their communication strategy based on performance requirements. However, message passing is the only directly available communication mechanism. Some modules use it to introduce shared state, e. g., for global configuration flags. If a communication requires any form of response, message passing is used consistently throughout the VM.

Interpreter instances being event-loops with private-by-default memory make their communication explicit. This enables us to determine easily which parts of the memory are shared and where potential data races can occur. While the modules use the global message passing mechanism, their particular communication protocol remain encapsulate inside a module. Thus, the modules can rely on their own communication strategies without interfering with each other.

4.2.2 Global Synchronization and Object Movement

Taking only the VM-related state into account is insufficient. Garbage collection (GC) and the support of explicit object movement require special care of all obtained direct memory addresses that could be invalidated while they are used.

A first precaution is the use of stop-the-world for all GC and object movement related operations. The synchronization is done with so-called *safepoints*. Safepoints are known states of the interpreter in which enough of the invariants are known to safely join a global request for synchronization. These safepoints are requested explicitly in the code for all operations that can lead to a call to the GC, inter-core allocation, or explicit object movement.

Accepting safepoints however, is done implicitly. Every message send waiting for an acknowledgment can potentially receive a safepoint message. Since arbitrary global safepoints are prone to deadlocks and not all code can handle the subsequent object movement, we have a mechanism that tracks for each interpreter instance whether it is able to accept a safepoint. This *safepoint ability* is an inherently cross-cutting concern, tied to the implementation specifics of every piece of code that can result in a message send. The safepoint ability implies that the C++-stack does not hold any references to objects in the heap, which could potentially move and thus would result in invalid references.

We either make sure that pointers are refetched/remapped after a potential GC, or if possible, disallow safepoints in a dynamic scope. This choice is a strict case-by-case decision, since it is a complex tradeoff between performance and deadlock safety. Refetching pointers results in higher memory overhead, but disabling safepoints can result in deadlocks and increased message response times.

Automating these decisions might be possible either with a sophisticated type-system, or as part of the infrastructure of a just-in-time compiler, or by changing the implementation language to a garbage-collected one to avoid the problem. Modularizing these decisions however, is not necessarily desirable since the decisions are inherently tied to the specific context in which they are applied.

4.2.3 Experiment: Threads vs. Processes

We ported the RoarVM from the TILE64 to commodity multicore systems running Linux and Mac OS X. Since both operating systems lack library and debugger support for process-based applications, we used threads instead.

The challenge therefore was to switch from a *private-by-default* memory model to a *public-by-default* one. This meant we needed to identify all state that previously used C++ language constructs like statics or globals and turn it into thread-local state.

Our inspection showed that only particular globals, especially the references to interpreter instance and memory system, were use throughout the whole system. Other globals were used only inside a particular modules, i. e., they were properly encapsulated. Furthermore, many globals in the RoarVM turned out to be written only once during initialization. Here, single-assignment variables with more flexibility than C++'s `const` would be valuable.

While the refactoring brought changes to most parts of the system, they were not necessarily related to modularity. The lexical visibility of globals is easily restricted so that they become private to a module. However, the execution semantics of threads, which is inherently orthogonal to modules, asks for additional language semantics like efficient thread-local state and single-assignment variables. Problematic language constructs are all kind of global state, especially static variables in C functions. They enable modular, and concise state, but have no place in multi-threaded code because of the data races they typically provoke.

4.2.4 Experiment: Work-Stealing Scheduler

Breaking the bottleneck of a single central scheduler is a change that also has cross-cutting impact on the code. The scheduling data structure needs to allow multiple instances, one for every core, and the scheduling strategy needs to change to be able to utilize the new capabilities, while maintaining the relevant semantics of the previous scheduling implementation. This results in changes to all parts of the system that interact with the scheduling.

In our experiment, we decided to remodularize part of the system to introduce an explicit scheduling module that maintains the most important invariants. The critical parts to get right here are the locking strategy and the work-stealing. Both parts require care to avoid deadlocks and situations like executing the same Smalltalk `process` multiple times. A naming convention that indicates whether a particular method acquires a lock, helped to avoid obvious deadlocks.

Better tool support would have been beneficial for the refactoring. From a language perspective however, C++'s abstractions were powerful enough. The new module often relies on existing modules for the actual implementation, while it itself concentrates on the concurrency aspects. The naming conventions would ideally be checked for consistency either by tooling or a language mechanism.

4.3 SLIP

The main drivers behind the modularization of SLIP's implementation are purpose and dominating invariants. In all successive versions of the SLIP interpreters, the memory module is the most prevailing one, since it underlies all aspects of the interpreter. Another important module is the thread module (not to be confused with multi-threading), which manages continuations in terms of a thread of frames forming an explicit stack.

Since memory management is the most cross-cutting aspect in SLIP, it was treated with special care. Together with the garbage collector, conventions were introduced to mark all functions that perform activities related to memory management. This naming convention makes it explicit where on-stack pointers to heap objects have to be refetched after a potential GC. While the RoarVM applied an approach that made distinctions for different cases, SLIP implementations apply a consistent refetch-always policy.

These are the most obvious cross-cutting concerns in SLIP, and our conclusion is similar to the one for RoarVM: the VM implementation could benefit from either being implemented in a garbage-collected language, removing the need for such conventions, or in a language that enforces the conventions of such suffixes and pointer refetching policy.

4.3.1 Experiment: Multi-Threading

SLIP version 14 adds support for the concurrency constructs `spawn` and `sync`. The special form `spawn` starts a `process`, i. e., a POSIX thread. Its companion procedure `sync` blocks on a `process` until that `process` has completed and returns its result. To this end, the `concurrency` module and the `context` module were added. The addition of the concurrency module, encapsulating implementation details of the operating system's threading library, did not impact the design of the existing implementation.

However, the context module is based on a redesign and remodularization of the system. The context module defines the execution context for each `process`, including the current expression or value, environment, frame, and the thread of frames. The `context` becomes the central element of execution and is required everywhere in the system. All calls to the thread module, environment module and memory manager module are first channeled through the context module. The context module ensures the relevant invariants—most notably the safepoint management—and will then dispatch the actual work to a more specific module.

We conclude that the complete remodularized with regard to the new dominating invariants resulted in a system with high consistency and maintainability. However, investing such an effort is not always possible, and it is also not clear how to trade off the different invariants to determine the dominating one in the general case. While language support can help, refactoring tools would be more beneficial.

5. Related Work

A similar study was done by Haupt et al. [5] regarding the general approach for modularization of virtual machines. They find that there is no single strategy to decompose multiple VMs. However, they find that VM services are frequently the main entities constituting modules. The study does not regard aspects of concurrency and therefore it remains unclear whether concurrency was guiding modularization.

Cantrill and Bonwick [2] discuss concurrency as a general concern of system software. They give advice on how to build complex systems. The two most important points of advice are (i) do synchronization on shared state into a single module, and (ii) avoid mutable shared state where possible. This coincides with our experience as well.

6. Conclusions

Our case study lead us to the conclusion that concurrency is one of the main drivers for modularization. Techniques like communicating event loops encourage designs with clear communication interfaces that avoid shared mutable state where possible. If shared state cannot be avoided, it should be internal to a module. Such a design simplifies reasoning about correctness and freedom of data races significantly.

We identified a couple of techniques that support such designs. The use of operating system *processes* makes shared memory explicit, raising awareness for sharing and potential data races. A notion of *ownership* would refine this further when mutation is restricted to owners, and ownership could be transferred. Support for *immutability*, *single assignment*, and *asynchronous message passing* are valuable language constructs, too. None of these techniques is directly related to modularization concepts; however, from our perspective these concepts would be the necessary foundation to enable better modularization of complex systems. They enable developers to contain mutation, and thus concurrency issues, inside of module boundaries.

Concurrency invariants are one of the strongest drivers behind modularization decisions. Often, they are the hardest part to *get right* and this inherent complexity can justify remodularization of a system to ensure correctness. This requires a language with expressive modularization constructs. However, it is even more essential to have good tools to support refactoring and remodularization when necessary.

The cross-cutting aspects we found most important are related to global invariants like synchronization and garbage collection. None of these seem to be amenable to modularization. We found them to be extremely tied to the code. They would require one-to-one mapping of aspects onto join points, which would reduce maintainability and increase the chance of subtle bugs. Instead, for these kind of problems, we expect better results from approaches that enable us to verify the consistency of applied conventions. Possible approaches could be smart annotations [6] or PyPy's application of a transformation pipeline [7].

Acknowledgments

Stefan Marr is supported by doctoral scholarships granted by IWT-Vlaanderen, Belgium. Jens Nicolay is funded by the *COGNAC* project sponsored by FWO Vlaanderen. Tom Van Cutsem is a post-doctoral fellow of FWO Vlaanderen.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[2] B. Cantrill and J. Bonwick. Real-world concurrency. *Commun. ACM*, 51(11):34–39, 2008.

[3] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. of OOPSLA'98*, 1998.

[4] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, pages 3–12, 2007.

[5] M. Haupt, B. Adams, S. Timbermont, C. Gibbs, Y. Coady, and R. Hirschfeld. Disentangling virtual machine architecture. *IET Software, Special Issue on Domain-Specific Aspect Languages*, 3(3):201–218, June 2009.

[6] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt. Co-evolving annotations and source code through smart annotations. In *Proc. of CSMR'10*. IEEE CS, 2010.

[7] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Proc. of OOPSLA'06*, pages 944–953, 2006.

[8] D. Ungar and S. S. Adams. Harnessing emergence for manycore programming: Early experience integrating ensembles, adverbs, and object-based inheritance. In *Proc. of SPLASH'10*, pages 19–26. ACM, 2010.

[9] D. Ungar and S. S. Adams. Hosting an object heap on manycore hardware: An exploration. In *Proc. of DLS'09*. ACM, 2009.