

Reliable Shared Memory Communication: A Position Statement on Synchronization and Composition

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen

Argentinierstraße 8, A-1040 Vienna, Austria

E-mail: franz@complang.tuwien.ac.at

Abstract

Much work was done on distributed shared memory and consistent shared memory models. However, communication through shared memory (read after write) is a low-level language-concept – too low for many programming tasks if not protected by a type system. We propose static types to enforce proper synchronization of shared memory communication in concurrent and even some sequential cases. For component composition this approach (and other type-based techniques) can probably profit from support of certification by the operating system.

1 Introduction

Today shared memory models seem to be predominant in concurrent programming. Even on distributed systems we apply complicated protocols to get the delusion of a (more or less uniform) shared memory. An important reason is obvious: We want to have a single programming model as abstraction over many different hardware architectures. On shared memory it is relatively easy to simulate other programming models.

Shared memory has its disadvantages as a high-level programming model: In concurrent systems, communication through shared variables (read after write) and synchronization are regarded as two different language concepts, and frequently programmers get synchronization wrong. We often use higher-level concepts like synchronized methods (Java terminology) in monitors, rendezvous communication, MPI, etc. where communication and synchronization are strongly related. There are approaches (like active objects) to eliminate direct shared memory communication, but they will not replace more conventional object-oriented programming techniques in the foreseeable future. Furthermore, shared memory communication can be synchronized in a wrong way even without concurrency. For example, also in purely sequential programs a producer can overwrite a buffer entry before a consumer has read the overwritten entry (although such problems can be found much easier in sequential than in concurrent programs).

In this paper we argue that static type checking can ensure proper synchronization of shared memory communication thus eliminating its most important disadvantage. Furthermore, we formulate wishes on operating systems concerning reliable shared memory communication and type-based techniques in general, especially wishes helpful in composition of software components supported by certificates.

2 Making Shared Memory Communication Reliable

Process types [6] allow us to specify constraints on the acceptability of messages (history properties [4]) in a type-safe way. The following simple buffer example in an experimental language shows how to use these types:

```
class Buffer is
  token empty, filled      -- declaration of tokens
  private var elem: Integer -- variable accessed by several threads
  method put (e: Integer; with empty->filled) do elem = e
  method get (with filled->empty): Integer do return elem
  creator new: Buffer[empty] do Null
```

According to the `with`-clause in `put` we can invoke `put` only if we have a token `empty`; this token is removed on invocation, and `filled` is added on return. For a variable `x` of type `Buffer[empty]` (a buffer with a single slot known to be empty) we can invoke `x.put(...)` changing the type of `x` to `Buffer[filled]`. Then we can invoke `x.get`, then again `x.put(...)`, and so on. The creator `new` issues only a single token for each instance. Since no `with`-clause changes the number of available tokens, there is always just a single token available. Therefore, all accesses to `elem` in `put` and `get` occur alternating in sequential ordering. In this case a compiler can statically ensure proper synchronization just by comparing the number of available tokens with the tokens required on method invocation. No further synchronization is necessary. However, each client must know the current buffer state before invoking `put` or `get`.

The example shows how we can base synchronization on the availability of tokens. To be practically useful we must extend the approach in several ways:

Internal Tokens: While tokens expressed in types (as in `Buffer[empty]`) must be statically known, tokens managed by the object itself can be changed and checked in statically unforeseeable cases. For example,

```
method put (e: Integer) when empty->filled do elem = e
```

dynamically checks whether the object has the needed token. Otherwise execution blocks until `empty` becomes available in the object. Then, `empty` is removed when entering the body, and `filled` is added at the end. It is easy to implement such semantics by a semaphore for each token name and invoking P for tokens to the left of `->` at the begin and V for tokens to the right at the end. This concept resembles monitors and is useful only in concurrent systems.

Bounds: In general, `with`-clauses and `when`-clauses can specify nearly arbitrary changes of tokens. Fortunately, we can always compute upper bounds on available token sets locally from class definitions. Variable accesses are well-synchronized if surrounding `with`-clauses and `when`-clauses of each two methods accessing a variable together contain to the left of `->` more tokens than can be available according to upper-bound sets. Hence, a rather inexpensive algorithm can ensure proper synchronization (without semaphore operations for `with`-clauses).

Read Access: Concurrent state-independent read-accesses need not be prevented.

Unchecked Sequential Code: For variables accessible only by a single thread we can explicitly switch off checking of proper synchronization. Today we know techniques for escape analysis that can determine many sequential cases [2, 8].

Several approaches to ensure reliable shared memory communication were proposed for concurrent systems [1]. Our approach using tokens is unique by considering also shared memory communication in sequential programs if programmers want to do so. Without annotations (like `with`-clauses) it is difficult or impossible for a compiler to figure out from the above example that `put` and `get` shall be invoked only in alternation. It is unlikely that approaches without annotations can be realized in a useful way. Furthermore, advanced escape analysis (often used in checking proper synchronization) needs access to the whole program code; there is no separate compilation. Each program change can affect necessary synchronization in seemingly unrelated program parts.

The approach based on tokens supports separate compilation since simple local escape analysis is sufficient, and program changes cannot affect seemingly unrelated parts.

If we can analyze a complete program as a whole (no separate compilation), then it is in most cases not necessary to annotate types with tokens, this is, we can simply write `Buffer` instead of `Buffer[empty]`. The needed tokens can be inferred statically from `with`-clauses [7]. This inference has not yet been explored for types of instance variables (which we expect to work) and together with genericity (a completely open problem).

Another open question is whether we can infer `with`-clauses from program code. In simple cases it is obviously possible to infer something. For example, if class `Buffer` (on the previous page) was defined without any `with`-clause, then we could infer from the use of variable `elem` that there must be `with`-clauses in `put` and `get`. The least restrictive clause would be ‘`with t->t`’ on both methods for a new token `t`. However, it is impossible to infer from this code that `put` and `get` shall be invoked in alternation; nothing in the code prevents us from reading a buffer entry several times and from overwriting it before it has been read. Probably it is only possible to infer from program code situations where methods must not be invoked simultaneously because they access the same variable, and all inferred clauses are of the form ‘`with t1, ..., tn->t1, ..., tn`’ where each `ti` is a token specifying access to a specific instance variable. Such `with`-clauses ensure sequential access to all methods except for methods accessing non-overlapping sets of variables. Therefore, inferred `with`-clauses give use just a little bit more freedom than switching off synchronization in sequential code. Explicit `with`-clauses have a different quality by specifying constraints not available otherwise. Likewise, `when`-clauses have a different quality than `synchronized` functions in Java because `when`-clauses cannot just be used to ensure mutual exclusion; they can also support limited concurrent execution (like atomic actions [3]) and substitute `wait` and `notify` operations.

3 Component Composition and Certificates

Only an object itself can issue tokens for this object. Static type checking prevents clients from arbitrarily introducing or replicating tokens. Unfortunately, current linking techniques do not ensure the required type safety of components linked together. As

long as nobody was interested in combining unchecked components, software engineering processes ensured proper checking. However, deliberately breaking type safety gives components a chance to access otherwise hidden resources. Hence, there is interest in providing malicious unchecked components and we need checks on linking.

Here are some approaches to prevent unchecked components from doing harm:

Direct Dynamic Check: Before performing an action (accessing a variable) we make sure the action is safe (acquire a lock anyway). This approach is not only expensive but also dangerous because of a probably wrong granularity of the atomic action. The intended behavior (`put` and `get` in alternation) cannot be guaranteed.

Indirect Dynamic Check: The run-time system remembers all tokens other components got and checks on method invocations (originating from other components) whether the needed tokens are actually available. This approach is expensive. It is safe although problems may be detected quite late.

Load-Time Check: We repeat type checks when linking components together. This approach is safe and detects problems early. Repeated type checks are expensive and require all information needed in type checks (annotations) to be available. Unfortunately, to make load-time checks easier it is necessary to provide code in a form that makes also re-engineering easy.

Proof-Carrying Code: Code comes together with correctness proofs [5]. We need not perform complex code checks at load time, but perform less expensive checks whether proofs are correct and correspond to the code at load or execution time. We can regard code with type annotations as a special kind of proof carrying code. There are approaches to encode types such that program executions fail without executing extra code for checks (through dynamic binding) if types are not correct. However, in our case these techniques are not easily applicable because types (especially token sets) change dynamically with each invocation.

Certificates: To avoid repeated type checks we give checked components a certificate. Inexpensive checks for the availability of required certificates replace complete type checks. We must make provision against faked certificates.

The approach using certificates is promising: It is reliable, efficient, and does not require all information needed in type checks to be available for component composition. Certificates are already in use especially in web applications. A single component can be associated with a large number of certificates for many different purposes, for example certificates saying “tested and certified by company X” (to improve the users’ trust in the component) or “conformance to interface I according to specification Y verified by tool Z” (no need to repeat this verification). The former kind of certificates is quite weak and essentially a marketing tool. It is of very limited use for our purpose because it does not give any concrete promises. The latter kind is not primarily intended to be seen by users, but supports reliable communication between tools. One tool can tell another tool that it has been applied to code and what it has found out about the code.

We regard certification rather as a task of an operating system than of language processing tools (mainly because the operating system has much better control over system communication, file access, program invocation, and so on) and envision following

scenario: A linker asking the operating system to open a file with object code specifies required certificates, and the operating system returns a handle to the file only if it has these certificates. The operating system adds certificates for a specific tool when requested by the tool only to files generated by the tool or at least read by successful tool applications. Furthermore, the operating system is responsible for removing old certificates from changed files and keeping valid certificates when copying and transmitting files. If a certificate depends on several files – for example, code is verified to conform to interfaces –, then the certificate must be regarded as no longer existent when any of the files containing code and interfaces change (no matter where the files are located). Of course, managing dependent certificates is much more difficult than just adding some bytes to generated web pages as is usually done today. Such operating system support would make component composition more reliable without the overhead of run-time or load-time checks and without need for any user interaction. It would work for data files as well as for executable code. This approach can be combined with proof-carrying code.

4 Conclusions

We have shown how type-based techniques can improve the reliability of communication through shared memory. In the authors opinion, code in current programming languages does not provide enough information to ensure proper synchronization of all shared memory communication especially together with separate compilation. We need annotations like those in the proposed `with`-clauses. As many type-based techniques this approach requires type checks on all components linked together. Automatically generated certificates can ensure that these checks have been successful. We consider the management of certificates as an operating system task.

References

- [1] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA 2001*. ACM, 2001.
- [2] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA '99*, Denver, Colorado, November 1999.
- [3] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '93*, pages 388–402, Anaheim, California, USA, October 2003. ACM.
- [4] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. Proceedings OOPSLA'93.
- [5] George C. Necula. Proof-carrying code. In *POPL '97*, January 1997.
- [6] Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, number 1241 in Lecture Notes in Computer Science, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.
- [7] Franz Puntigam. State inference for dynamically changing interfaces. *Computer Languages*, 27:163–202, 2001.
- [8] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI '03*, pages 115–128. ACM Press, 2003.