libcppa – Designing an Actor Semantic for C++11

Dominik Charousset
HAW Hamburg
dcharousset@acm.org

Thomas C. Schmidt
HAW Hamburg
t.schmidt@ieee.org

ABSTRACT

Parallel hardware makes concurrency mandatory for efficient program execution. However, writing concurrent software is both challenging and error-prone. C++11 provides standard facilities for multiprogramming, such as atomic operations with acquire/release semantics and RAII mutex locking, but these primitives remain too low-level. Using them both correctly and efficiently still requires expert knowledge and hand-crafting. The actor model replaces implicit communication by sharing with an explicit message passing mechanism. It applies to concurrency as well as distribution, and a lightweight actor model implementation that schedules all actors in a properly pre-dimensioned thread pool can outperform equivalent thread-based applications. However, the actor model did not enter the domain of native programming languages yet besides vendor-specific island solutions. With the open source library libcppa, we want to combine the ability to build reliable and distributed systems provided by the actor model with the performance and resource-efficiency of C++11.

Categories and Subject Descriptors

D.3.3 [**Programming languages**]: Language Constructs and Features—*Concurrent programming structures*

Keywords

C++, actor model, pattern matching

1. INTRODUCTION

The actor model is a formalism describing concurrent entities, "actors", that communicate by asynchronous message passing [13]. An actor can send messages to addresses of other actors and can create new actors. Actors do not share state and are executed concurrently.

Because Actors are self-contained and do not rely on shared resources, race conditions are avoided by design. The message passing communication style also allows network transparency and thus applies to both concurrency, if actors run on the same host on different cores/processors, and distribution, if actors run on different hosts connected via the network.

The actor model inspired several implementations in computer science, either as basis for languages like Erlang or as libraries/frameworks such as the Scala Actor Library to ease development of concurrent software. However, low-level primitives are still widely used in C++. The standardization committee added threading facilities and synchronization primitives to the C++11 standard [15], but did neither address the general issues of multiprogramming, nor distribution.

This work targets at concurrency and distribution in C++. We design and implement a library for C++, called libcppa, with an internal Domain-Specific Language (DSL) approach to provide high-level abstraction.

In the remainder, we discuss the actor model and related work in Section 2. Section 3 details design decisions of our internal domain-specific language approach. The pattern matching implementation of libcppa, which is an important ingredient in our library design, is presented in Section 4. Section 5 discusses message passing in libcppa. In Section 6, we present our algorithm for message queueing, which is a critical component of actor systems. Section 7 evaluates the performance characteristics of libcppa. Section 8 discusses limitations we had to face in C++11 when implementing libcppa. Finally, we report on future work in Section 9.

2. BACKGROUND AND RELATED WORK

Concurrent software components often need to share or exchange data. In shared memory environments, this communication happens implicitly via shared states. However, unsynchronized parallel access to shared memory segments easily leads to erroneous behavior caused by race conditions. Synchronization protocols are usually built on low-level primitives such as locks and condition variables that prevent parallel or interleaved execution of so called *critical sections*. This is inherently error-prone and correctly implementing critical sections requires expert knowledge, about compiler optimizations and out-of-order execution for instance [16]. Still, designing a suitable locking strategy is not the only challenge developers face on parallel hardware. Applications with good performance on a uniprocessor machine may experience a performance degradation on multiprocessor platforms, e.g., due to *false sharing*. False sharing occurs whenever two or more processors are repeatedly writing into a memory region that is mapped to more than one processor cache. This mutually invalidates the caches, even

if the processors are accessing distinct data, and seriously slows down execution [21]. An object that needs to support parallel access by a group of participants should not rely on locking, since a critical section is a performance bottleneck per se. Lock- and wait-free algorithms [11] scale in a multi-processor environment, but are significantly more complex to implement and verify [6].

2.1 The Actor Model

Higher-level abstractions following message passing or trans-actional memory paradigms were developed to establish pro-gramming models that are free of race conditions, and thus do not require synchronization of readers and writers. Trans-actional memory can be implemented in hardware [12] or software [19], and can be seamlessly integrated into new programming languages, as the example of Clojure [10] il-lustrates. Transactional memory scales for multiprocessor machines and has been implemented for tightly-coupled dis-tributed systems such as clusters [4], but it applies neither to communication between heterogeneous hardware compo-nents, nor to widely distributed applications. Message pass-ing, on the other hand, has proven to scale in multiprocessor environments as well as in both tightly and loosely coupled distributed computing. In the high-performance domain, message passing systems based on the MPI are used for decades [7].

The actor model [13] is build on the message passing paradigm, but raises the level of abstraction even further. It not only describes how software components communicate, but also characterizes the communicating components, the actors themselves. Based on this model, concurrent and dis-tributed systems can be composed of independent modules [1] that are open for communication with external compo-nents [2]. In addition, the actor model addresses reliability and fault tolerance in a network-transparent way [3].

2.2 Message Processing

The original actor modeling of Agha [1] introduced mailbox-based message processing. A mailbox is a FIFO ordered message buffer that is only readable by the actor owning it, while all actors are allowed to enqueue a new message. Since actors do not share state, the mailbox is the only way of communication among actors. Implementation concepts of mailbox management divide into two categories.

In the first category, an actor iterates over messages in its mailbox. On each receive call, it begins with the first but is free to skip messages. This postponing of messages can be automated, if the actor's behavior is defined as a partial function [9]. As actors can change their behavior in response to a message, the newly defined behavior may then be applied to previously skipped messages. A message remains in the mailbox until it is eventually processed and removed as part of its consumption. Actor systems that are based on this type of message processing include Erlang [3] and the Scala Actors library [9].

The second category of actor systems restricts the op-erations of message processing. In these approaches, the runtime system invokes a message handler exactly once per message in order of arrival. Some systems allow changing the message handler, but an untreated message cannot be recaptured later. Examples of this kind are SALSA [23], Akka [22], Kilim [20], and Retlang [18].

We have followed the first approach, since this allows ac-tors to prioritize messages and to wait for a response prior to returning to the default behavior. Furthermore, pattern matching has proven useful and very effective to ease defi-nition of partial functions used as message handlers [8]. We provide pattern matching for message handling as a domain-specific language in our system.

2.3 Fault Propagation

Actors achieve fault tolerance by monitoring each other [13], which is necessary in particular for constructing fault-tolerant distributed systems. Whenever an actor fails, an exit message is sent to all actors that monitor it. This mon-itoring can be bidirectional, and actors with such a strong relation are called *linked*. Linked actors form a subsystem in which errors are propagated via exit messages.

Based on this mechanism, developers can build subsys-tems in which all actors are either alive or have collectively failed. Each subsystem can include one or more actors that survey working actors and re-create failing workers. In pro-ceeding this way, hierarchical, fault-tolerant systems such as Erlang's *supervision trees* [3] can be built. Newer imple-mentations of the actor model (e.g., Kilim, Akka, and the Scala Actors Library) have adopted Erlang's model of error propagation, as it has proven very effective, elegant and reli-able [17]. It was therefore natural for us to adopted Erlang's well-established fault propagation model as well.

3. ACTOR SEMANTIC AS INTERNAL DSL

Our aim is to add an actor semantic to C++ that en-ables developers to build efficient concurrent and distributed software. Though `libcppa` is influenced by functional pro-gramming languages such as Erlang and Scala, it should not mimic a functional programming style and provide an API that looks familiar to C++ developers.

We decided to use an internal Domain-Specific Language (DSL) approach to raise the level of abstraction in C++, while requiring as little glue code as possible. An essential ingredient of our design is the keyword-like identifier `self`. From a user's pointer of view, it identifies an actor similar to the implicit `this` pointer identifying an object within a member function.

The interface for actors is essentially split in two. The first interface, called `actor`, provides all operations needed for linking, monitoring, and delivering messages. In `libcppa`, actors are always guarded by `actor_ptr`, which is a smart pointer type. The second interface is `local_actor`. It rep-resents the *private* access to an actor, since only the actor itself is allowed to dequeue a message from its mailbox. The keyword `self` behaves like a pointer of type `local_actor*`.

3.1 Cooperative Scheduling of Actors

Scalability in the context of multi-core processors requires to split the application logic into many independent tasks that could be executed in parallel. An actor is a represen-tation of such an independent task. This makes lightweight creation and destruction of actors mandatory. It is not fea-sible to map each actor to its own thread, since creation and destruction of threads is heavyweight. Thread manage-ment relies on system calls and acquires system resources such as thread state in the OS scheduler, stack and signal stack. Thus, short-living threads do not scale well, since the effort for creation and destruction outweighs the benefit of parallelization.

An ideal way to ensure *fairness* in an actor system requires preemptive scheduling. A fair system would guarantee that no actor could starve other actors by occupying system resources. However, unrestricted access to hardware or kernel space is needed to implement preemptive scheduling, since it relies on hardware interrupts to switch between two running tasks. No operating system can allow unrestricted hardware or kernel space access to a user space application for obvious reasons.

In general, user space schedulers can only implement cooperative scheduling, i.e., execute actors in a thread pool. But there is some design space to make context switching implicit. The two operations each actor uses frequently are sending and receiving of messages. Thus, the library could switch the actor's context back to the scheduler, whenever it sends or receives a message. An ordinary workflow of an actor is receiving messages in a loop and sending messages either as part of its message processing or after computing results. Thus, interrupting an actor during send will probably interrupt an actor during its message processing, while interrupting it during receive seems natural in this workflow. Furthermore, we need to support interruption during receive since an actor is not allowed to block while its mailbox is empty. Instead, an actor returns to scheduler and is re-scheduled again after a new message arrives to its mailbox.

Nevertheless, developers can choose to opt-out of the cooperative scheduling and execute an actor in its own thread. This is particularly useful, whenever an actors needs to call blocking functions and therefore would possibly starve other actors in a cooperative scheduling.

3.2 Context-Switching Actors

Context-switching actor use so called *fibers* or *lightweight threads*. Like a kernel thread, each fiber has its own stack. However, fibers are scheduled in user space only, and are scheduled on top of kernel threads. Hence, the ratio between fibers and threads is N:M, as any number of fibers can be cooperatively scheduled on top of kernel threads.

We provide this actor implementation to ease soft migration strategies, where a previously threaded application is ported to actors. The API for context-switching actors is equal to the API for threaded actors and provides a 'blocking' – at least from the user's point of view – receive function. However, this implementation does not scale up to large actor system. As an example for a current mainstream system: Mac OS X defines the recommended stack size to 131,072 bytes, and the minimally allowed stack size to 32,768 bytes. Assuming a system with 500,000 actors, one would require a memory usage of *at least* 15 GB of RAM for stack space only. This would rise up to 61 GB with the recommended stack size instead in use. This clearly does not scale well for large systems. To reduce memory usage and scale up to large systems, we provide a third, event-based approach.

3.3 Event-Based Actors

A common problem of event-based APIs is *inversion of control*. Event-based APIs usually define a callback that is invoked for each event. However, callback-based message processing cannot prioritize messages since the callback is invoked by the runtime system usually in arrival order, and thus has a semantic different from our mailbox-based actors using the receive statement. Therefore, we use a behavior-based API with mailbox-based message processing.

An event-based actor sets its required behavior as a partial function using the `become` function. This partial function is used until the actor replaces it by invoking `become` again. Thus, all actor implementations in `libcppa` are able to prioritize communication.

3.4 Emulating the Keyword "self"

An actor semantic needs to be consistent. For that reason, we introduce the identifier `self`, which is not allowed to be invalid or to return `nullptr`. Otherwise, it could not be guaranteed that receive or send statements never fail. This implies that a non-actor caller, i.e., a thread, is converted to an actor if needed. For instance, when using send and receive functions in `main`, the corresponding thread should be converted implicitly to an actor.

Unlike `this`, `self` is not limited to a particular scope. Furthermore, it is not just a pointer, but it needs to perform implicit conversions on demand. Consequently, `self` requires a type allowing implicit conversion to `local_actor*`, where the conversion function returns a thread-local pointer. Our approach shown below uses a global `constexpr` variable with a type that behaves like a pointer.

```
class self_type {
  static local_actor* get_impl();
  static void set_impl(local_actor* ptr);
 public:
  constexpr self_type() { }
  inline operator local_actor*() const {
    return get_impl();
  }
  inline local_actor* operator->() const {
    return get_impl();
  }
  inline void set(local_actor* ptr) const {
    set_impl(ptr);
  }
};
namespace { constexpr self_type self; }
```

The `constexpr` variable `self` provides access to the implicit conversion operator as well as the dereference operator "`->`". From a user's point of view, `self` is not distinguishable from a pointer of type `local_actor`. The static member functions are implemented as follows.

```
thread_local local_actor* t_self = nullptr;
local_actor* self_type::get_impl() {
  if (!t_self) t_self = convert_thread();
  return t_self;
}
void self_type::set_impl(local_actor* ptr) {
  t_self = ptr;
}
```

Our approach adds little, if any, overhead to an application. In fact, `self` is nothing but syntactic sugar and the compiler could easily optimize away the overhead of using member functions. A `constexpr` variable does not cause a dynamic initialization at runtime, why the global variable `self` does not cause any overhead since it provides an empty `constexpr` constructor. Furthermore, all member functions are declared `inline`, allowing the compiler to replace each occurrence of a member function with a call to `self_type::get_impl`. `self`.set() is intended for in-library use only. The latter is needed to implement cooperative scheduling.

3.5 Copy-On-Write Tuples

A message can be send to multiple receivers or forwarded from one actor to another. In `libcppa`, messages always follow call-by-value semantic. This unburdens the developers of managing tuple lifetimes and keeps the programming model clean and easy to understand. However, sending a message to multiple actors would require multiple copies of the message, wasting both computing time and memory. We use a copy-on-write implementation to avoid both unnecessary copies and race conditions.

All messages use the type `any_tuple`, which represents a tuple of arbitrary length with dynamically typed elements. To restore the static type informations, `tuple_cast` can be used, e.g., `auto x = tuple_cast<int,int>(tup)` tries to cast `tup` of type `any_tuple` to a tuple of two integers and returns `option<cow_tuple<int,int>>`. Since the cast returns an option rather than a straight value, the user has to verify the result, just as users of `dynamic_cast` are required to do. The template class `cow_tuple<int,int>` then provides two access functions, `get` and `get_ref`. Unlike the access for `std::tuple`, we do not provide const overloaded functions. The reason for avoiding this is that non-const access has implicit costs attached to it, because the non-const access causes a deep copy operation if there is more than one reference to the tuple. Hence, we designed an explicit non-const access function, `get_ref`, to prevent users from accidently create unnecessary copies. Still, users will very seldom, if ever, interact with tuples directly, because the tuple handling is usually hidden by pattern matching, as shown in Section 4.

3.6 Spawning Actors

Actors are created using the function `spawn`. The recommended way to implement both context-switching and thread-mapped actors is to use functors, such as free functions or lambda expressions. The arguments to the functor are passed to `spawn` as additional arguments. The optional `scheduling_hint` template parameter of `spawn` decides whether an actor should run in its own thread or use context-switching. The flag `detached` causes `spawn` to create a thread-mapped actor, whereas `scheduled`, the default flag, causes it to create a context-switching actor. The function `spawn` is used quite similar to `std::thread`, as shown in the examples below.

```
#include "cppa/cppa.hpp"
using namespace cppa;

void fun1();
void fun2(int arg1, const std::string& arg2);

class dummy : public event_based_actor {
  // ...
  dummy() { /*...*/ }
  dummy(int i) { /*...*/ }
};

int main() {
  // spawn context-switching actors
  // equal to spawn<scheduled>(fun1)
  auto a1 = spawn(fun1);
  auto a2 = spawn(fun2, 42, "hello actor");
  // spawn a lambda expression
  auto a3 = spawn([]() { /*...*/ });
  auto a4 = spawn([](int) { /*...*/ }, 42);
  // spawn thread-mapped actors
  auto a5 = spawn<detached>(fun1);
  auto a6 = spawn<detached>(/*...*/);
```

```
  // spawn actors using class defintions
  auto a7 = spawn<dummy>();
  auto a7 = spawn<dummy>(42);
}
```

In general, context-switching and thread-mapped actors are intended to ease migration of existing applications or to implement managing actors on-the-fly using lambda expressions. Class-based actors usually subtype `event_based_actor`.

It is worth noting that `spawn(fun, arg0, ...)` is *not* the same as `spawn(std::bind(fun, arg0, ...))`. The function `spawn` evaluates `self` arguments immediately and forwards them as `actor_ptr` instances. When using `bind`, `self` is evaluated upon function invocation, thus pointing to the spawned actor itself rather than to the actor that created it.

4. PATTERN MATCHING IN C++

C++ does not provide pattern matching facilities. A general pattern matching solution for arbitrary data structures would require a language extension. Hence, we had decided to restrict our implementation to dynamically typed tuples, to be able to use an internal domain-specific language (DSL) approach.

4.1 Match Expressions

A match expression in `libcppa` begins with a call to the function `on`, which returns an intermediate object providing the member functions `when` and the right shift operator. The right-hand side of the operator denotes a callback, usually a lambda expression that should be invoked on a match, as shown in the examples below.

```
on<int>() >> [](int i)
on<int,float>() >> [](int i, float f)
on<int,int,int>() >> [](int a, int b, int c)
```

The result of such an expression is a partial function that is defined for the types given to `on`. A comma separated list of partial functions results in a single partial function that sequentially evaluates its subfunctions. A partial function invokes at most one callback, since the evaluation stops at the first match.

```
auto fun = (
  on<int>() >> [](int i) { /*case1*/ },
  on<int>() >> [](int i) {
    // unreachable; case1 always matches first
  }
);
```

Due to the C++11 grammar, a list of partial function definitions must be enclosed in brackets if assigned to a variable. Otherwise, the compiler assumes commas to separate variable definitions.

The function "`on`" can be used in exactly two ways. Either with template parameters only or with function parameters only. The latter version deduces all types from its arguments and matches for both type and value. The template "`val`" can assist to match for types.

```
on(42) >> [](int i) { assert(i == 42); }
on("hello world") >> []() {}
on(1, val<int>) >> [](int i) {}
```

Callbacks can have less arguments than given to the pattern, but it is only allowed to skip arguments from left to right.

```
on<int,int,float>() >> [](float)
on<int,int,float>() >> [](int, float)
on<int,int,float>() >> [](int, int, float)

// invalid: on<int,int,float>() >> [](int i)
```

4.2 Atoms

Assume an actor provides a mathematical service on integers. It takes two arguments, performs a predefined operation and returns the result. It cannot determine an operation, such as *multiply* or *add*, by receiving two operands alone. Thus, the operation must be encoded into the message. The Erlang programming language introduced an approach to use non-numerical constants, so-called *atoms*, which have an unambiguous, special-purpose type and do not have the runtime overhead of string constants. Atoms are mapped to integer values at compile time in `libcppa`. This mapping is collision-free and invertible, but limits atom literals to ten characters and prohibits special characters. Legal characters are "`_0-9A-Za-z`" and whitespaces. Although user-defined literals are the natural candidate to choose, we have implemented atoms using a `constexpr` function, called `atom`, because user-defined literals are not yet available at mainstream compilers.

```
on<atom("add"),int,int>() >> ...
on<atom("multiply"),int,int>() >> ...
```

Our implementation can only enforce the length requirement, but cannot enforce valid characters at compile time. Each invalid character is mapped to the whitespace character, why the assertion `atom("!?")` != `atom("?!")` is not true. However, this issue will fade away after user-defined literals become available in mainstream compilers, because it is then possible to raise a compiler error for invalid characters.

4.3 Reducing Redundancy

In our previous examples, we always have repeated the types from the left side of a match expression. To avoid such redundancy, `arg_match` can be used as last argument to the function `on`. This causes the compiler to deduce all further types from the signature of the given callback.

```
on<atom("add"),int,int>() >> [](int a, int b)
on(atom("add"),arg_match) >> [](int a, int b)
```

The second example calls `on` without template parameters, but is equal to the first one. When used, `arg_match` must be passed as last parameter. If all types should be deduced from the callback signature, `on_arg_match` can be used, which is equal to `on(arg_match)`. Both `arg_match` and `on_arg_match` are `constexpr` variables.

4.4 Wildcards

The type `anything` can be used as wildcard to match any number of any types. A pattern created by `on<anything>()` – or its alias `others()` – is useful to define a default "catch all" case. For patterns defined without template parameters, the `constexpr` value `any_vals` can be used as function argument. The constant `any_vals` is of type `anything` and is nothing but syntactic sugar for defining patterns.

```
on<int,anything>() >> [](int i)
  { /* tuple with int as first element */ },
on(any_vals,arg_match) >> [](int i)
  { /* tuple with int as last element */ },
others() >> [] { /* default handler */ }
```

4.5 Guards

Guards allow to constrain a given match statement by using placeholders, as the following example illustrates.

```
// contains _x1 - _x9
using namespace cppa::placeholders;

on<int>().when(_x1%2==0) >> []{ /* even */ },
on<int>() >> []{ /* odd */ }
```

Guard expressions are a lazy evaluation technique. The placeholder `_x1` is substituted with the first value of a given tuple. To reference variables of the enclosing scope, or member variables of the actor itself, we provide two functions designed to be used in guard expressions: `gref` ("guard reference") and `gcall` ("guard function call"). The function `gref` creates a reference wrapper that forces lazy evaluation.

```
int val = 42;
// (1) matches if _x1 == 42
on<int>().when(_x1 == val)
// (2) matches if _x1 == val
on<int>().when(_x1 == gref(val))
// (3) matches as long as val == 42
others().when(gref(val) == 42)
// (4) ok, because _x1 forces lazy evaluation
on<int>().when(_x1 == std::ref(val))
// (5) compiler error due to eager evaluation
others().when(std::ref(val) == 42)
```

Statement (5) in the example above is evaluated immediately and returns a boolean instead of a guard expression. The second function – `gcall` – encapsulates a function call. Its use is similar to `std::bind`, but there is also a short version for unary functions: `_x1(fun)` is equal to `gcall(fun, _x1)`.

```
typedef std::vector<int> ivec;
auto vsorted = [](const ivec& v) {
  return std::is_sorted(v.begin(), v.end());
};
on<ivec>().when(gcall(vsorted, _x1))
// equal to
on<ivec>().when(_x1(vsorted)))
```

4.6 Projections and Extractors

Projections perform type conversions or extract data from tuples. Functors passed for conversion or extraction operations must be free of side-effects and shall in partiular not throw exceptions, because the invocation of the function is part of the pattern matching process.

```
auto f= [](const string& str)-> option<int> {
  char* p = nullptr;
  auto result = strtol(str.c_str(), &p, 10);
  if (p && *p == '\0') return result;
  return {};
};
receive (
  on(f) >> [](int i) {
    // case 1, conversion successful
  },
  on_arg_match >> [](const string& str) {
    // case 2, str is not an integer
  }
);
```

The lambda function `f` is a `string` $\Rightarrow$ `int` projection, but rather than returning an integer, it returns `option<int>`. An empty `option` indicates that a value does not have a valid

mapping to an integer. Functors used as projection or extration must take exactly one argument and must return a value. The types for the pattern are deduced from the functor's signature. If the functor returns an `option<T>`, then `T` is deduced.

Our DSL-based approach to pattern matching has more syntactic noise than a native support within the languages itself, i.e., compared to functional programming languages such as Haskell or Erlang. However, our approach uses only ISO C++11 facilities, does not rely on brittle macro definitions, and after all adds little -if any- runtime overhead by using expression templates [24].

5. MESSAGE PASSING

By supporting thread-like and event-based actors, `libcppa` has to support blocking as well as asynchronous operations. By using a behavior-based approach, we are able to provide an asynchronous API without inversion of control.

5.1 Receiving Messages

We provide a blocking API to receive messages for threaded and context-switching actors. The blocking function `receive` sequentially iterates over all elements in the mailbox beginning with the first. It takes a partial function that is applied to the elements in the mailbox until an element was matched. An actor calling `receive` is blocked until it successfully dequeued a message from its mailbox or an optional timeout occurs.

```
receive (
  on... >> // ...
  after(std::chrono::seconds(1)) >> // timeout
);
```

The code snippet above illustrates the use of `receive`. Note that the partial function passed to `receive` is a temporary object at runtime. Hence, using receive inside a loop would cause creation of a new partial function on each iteration. To avoid re-creation of temporal objects per loop iteration, `libcppa` provides three predefined receive loops, `receive_loop`, `receive_for`, and `do_receive(...).until`, to provide a more efficient but yet convenient way of defining receive loops.

Due to the callback-based nature of event-based message handling, we have to provide a non-blocking API, too. Hence, we provide a behavior-based API, which enables event-like message handling without inversion of control [8].

An event-based actor uses `become` to set its behavior. The given behavior is then executed until it is replaced by another call to `become` or the actor finishes execution. Class-based actor definitions simply subtype `event_based_actor` and must implement the pure virtual member function `init`. An implementation of `init` shall set an initial behavior by using `become`.

```
struct printer : event_based_actor {
  void init() { become (
    others() >> [] {
      cout << to_string(self->last_received())
           << endl;
    }
  ); }
};
```

Another way to implement event-based actors is provided by the class `sb_actor` ("State-Based Actor"). This base class

calls `become(init_state)` in its `init` member function. Hence, a subclass must only provide a member of type `behavior` named `init_state`.

```
struct printer : sb_actor<printer> {
  behavior init_state = (
    others() >> [] {
      cout << to_string(self->last_received())
           << endl;
    }
  );
};
```

Note that `sb_actor` uses the Curiously Recurring Template Pattern [5], why the derived class must be given as template parameter. This technique allows `sb_actor` to access the `init_state` member of a derived class.

The following example illustrates a more advanced state-based actor that implements a stack with a fixed maximum number of elements. Since this example uses non-static member initialization, it might not compile with some currently available compilers.

```
struct fixed_stack : sb_actor<fixed_stack> {
  static constexpr size_t max_size = 10;
  std::vector<int> data;
  behavior empty = (
    on(atom("push"), arg_match) >>
    [=](int what) {
      data.push_back(what);
      become(filled);
    },
    on(atom("pop")) >> [=]() {
      reply(atom("failure"));
    }
  );
  behavior filled = (
    on(atom("push"), arg_match) >>
    [=](int what) {
      data.push_back(what);
      if (data.size() == max_size)
        become(full);
    },
    on(atom("pop")) >> [=]() {
      reply(atom("ok"), data.back());
      data.pop_back();
      if (data.empty()) become(empty);
    }
  );
  behavior full = (
    on<atom("push"),int>() >> []() {
      // discard value
    },
    on(atom("pop")) >> [=]() {
      reply(atom("ok"), data.back());
      data.pop_back();
      become(filled);
    }
  );
  behavior& init_state = empty;
};
```

Nesting receives in an event-based actor is slightly more difficult compared to context-switching or thread-mapped actors, since `become` does not block. An actor has to set a new behavior calling `become` with the `keep_behavior` policy to wait for the required message and then return to the previous behavior by using `unbecome`, as shown in the example below. An event-based actor finishes execution with normal exit reason if the behavior stack is empty after calling `unbecome`. The default policy of `become` is `discard_behavior` that causes

an actor to override its current behavior. The policy flag must be the first argument of `become`.

```
// receives {int, float} sequences
struct testee : sb_actor<testee> {
  behavior init_state = (
    on<int>() >> [=](int value1) {
      become (
        // the keep_behavior policy stores
        // the current behavior on the
        // behavior stack to be able to
        // return to this behavior later on
        // by calling unbecome()
        keep_behavior,
        on<float>() >> [=](float value2) {
          cout << value1 << " => "
               << value2 << endl;
          unbecome();
        }
      );
    }
  );
};
```

5.2 Sending Messages

The default way of passing a message to another actor is provided by the function `send`, which models asynchronous communication. However, we also provide a `sync_send` function that returns a handle to the response message. Synchronous response messages can be received using this handle only and are not visible otherwise by receive operations. This allows actors to identify response messages unambiguously. Furthermore, whenever receiving of a synchronous response message times out, the message is dropped even if it is received later on.

The functions `receive_response` and `handle_response` can be used to receive response messages, as shown in the following example.

```
// replies to atom("get") with a string
auto testee = spawn<testee_impl>();
// blocking API
auto future = sync_send(testee, atom("get"));
receive_response (future) (
  on_arg_match >> [&](const string& str) {
    // handle str
  },
  after(chrono::seconds(30)) >> [&]() {
    // handle error
  }
);
// event-based actor API (nonblocking)
auto future = sync_send(testee, atom("get"));
handle_response (future) (
  on_arg_match >> [=](const string& str) {
    // handle str
  },
  after(chrono::seconds(30)) >> [=]() {
    // handle error
  }
);
```

The function `receive_response` is similar to `receive`, i.e., it blocks the calling actor until either a response message arrived or a timeout occures.

Similar to `become`, the function `handle_response` is part of the event-based API and is used as "one-shot handler". The behavior passed to `handle_response` is executed *once* and the actor automatically returns to its previous behavior afterwards. It is possible to 'stack' multiple `handle_response`

calls. Each response handler is executed once and then automatically discarded.

In both cases, the behavior definition of the response handler requires a timeout, because a non-replying actor would always cause a deadlock otherwise.

Often times, an actor sends a synchronous message and then wants to wait for the response immediately afterwards. In this case, using either `handle_response` or `receive_response` is quite verbose. Therefore, the returned handle provides the two member functions `then` and `await`. Using `then` is equal to using `handle_response`, wheres `await` corresponds to `receive_response`, as illustrated by the following example.

```
// blocking API
sync_send(testee, atom("get")).await(
  on_arg_match >> [&](const string& str) {
    // handle str
  },
  after(chrono::seconds(30)) >> [&]() {
    // handle error
  }
);
// event-based actor API (nonblocking)
sync_send(testee, atom("get")).then(
  on_arg_match >> [=](const string& str) {
    // handle str
  },
  after(chrono::seconds(30)) >> [=]() {
    // handle error
  }
);
```
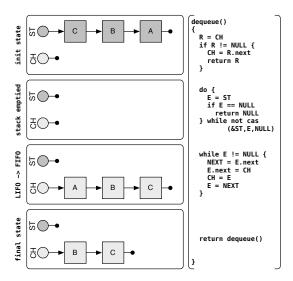
6. MAILBOX IMPLEMENTATION

Figure 1: Dequeue operation in a cached stack (ST = Stack Tail, CH = Cache Head)

The message queue or *mailbox* implementation is a critical component of any message passing systems. All messages sent to an actor are delivered to its mailbox, which acts as a shared resource whenever an actor receives messages from multiple senders in parallel. Thus, the overall system performance, foremost its scalability depends significantly on the selected algorithm.

A mailbox is a single-reader-many-writer queue. Everyone is allowed to enqueue a message to a mailbox, but only the
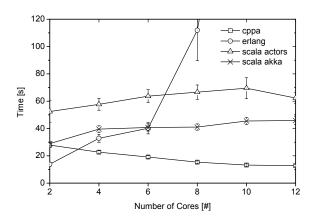
Figure 2: Mailbox performance in N:1 communication scenario

owning actor is allowed to dequeue a message. Hence, the dequeue operation does not need to support parallel access. We have combined a lock-free stack implementation with a FIFO ordered queue as internal cache. A lock-free stack can be implemented using a single atomic compare-and-swap (CAS) operation. It does not suffer from the so called *ABA* problem of concurrent access that can corrupt states in CAS-based systems [14] as the enqueue operation only needs to manipulate the *tail* pointer. However, without reordering the dequeue operation would have to traverse the (LIFO-sorted) stack in order to find the oldest element.

Figure 6 shows the dequeue operation of our mailbox implementation. It always dequeues elements from the FIFO ordered cache (CH). The stack (ST) is emptied and its elements are moved in reverse order to the cache, after the cache was drained. Emptying the stack can be done by a single CAS operation as it only needs to set ST to NULL.

Our mailbox has complexity $O(1)$ for enqueue operations while the dequeue operation has an average of $O(1)$ but a worst case of $O(n)$, where n is the current number of messages in the stack. Concurrent access to the cached stack is reduced to a minimum and both enqueueing and dequeueing perform only a single CAS operation.

6.1 Performance for N:1 Communication

To measure the efficiency of our mailbox implementation, we compare the runtime behavior of our software with common implementations of the actor model. Erlang and Scala are currently the most relevant languages for actor programming. We use their implementation as reference for concurrent computation. For Scala, we consider its standard library as well as the well-established third party library Akka [22]. In detail, our benchmarks are based on the following implementations of the actor model.

cppa `libcppa` applying event-based message processing.

erlang Erlang in version 5.9.2.

scala actors Scala with the event-based actor implementation of the standard library.

scala akka Scala with the Akka library in version 2.0.3

`libcppa` has been compiled with optimization level O4 of GNU C++ compiler version 4.7.2. Scala version 2.9.1 runs on a JVM configured with 4 GB of RAM.

We used 20 threads sending 1,000,000 messages each, except for Erlang, which has no threading library. In Erlang, we spawned 20 actors instead. The minimal runtime of this benchmark is the time the receiving actor needs to process the 20,000,000 messages and the overhead of passing the messages to the mailbox. More hardware concurrency leads to higher synchronization between the sending threads, since the mailbox acts as a shared resource. Furthermore, the workers in the thread pool are synchronized by a job queue in the implementations using a cooperative scheduling, which can be a concurrency bottleneck as well if actors perform short tasks and often need re-scheduling.

Figure 6.1 visualizes the time needed for the application to send and process the 20,000,000 messages as a function of available CPU cores. The ideal behavior is a decreasing curve, which reaches a global minimum given by the time the receiving actor needs to consume all messages one by one. For visibility reasons, we cut the y-axis at 120 s.
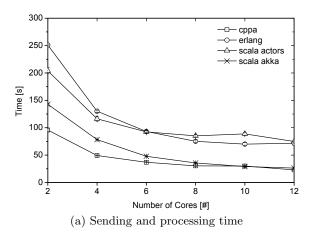
The processing overhead increases significantly for Erlang in case of more than 6 cores and directly correlates with the degree of concurrency. On average, it requires up to 700 seconds on 12 cores. `libcppa` attains the best behavior – a constantly decreasing curve – as its performance gain on additional hardware concurrency clearly outweighs the increasing synchronization overhead. Akka slightly outperforms Scala Actors, but has a steadily increasing curve, whereas Scala Actors reaches it maximum on ten cores and accelerates again on twelve cores. The results indicate that our mailbox implementation using the cached stack algorithm as well as our synchronization protocol among worker threads scale very well.

7. MIXED USE CASE PERFORMANCE

In this section, we consider a realistic use case including a mixture of operations under severe work load to evaluate the performance of `libcppa`. We again used the setup described in Section 6.1 for this benchmark, but also measure the memory consumption to examine both runtime and resource efficiency. The program creates a simple multi-ring topology with a fixed number of actors per ring. A token with an initial value of 10,000 is passed along the ring and decremented each round. A client that receives the token forwards it to its neighbor and terminates whenever the value of the token is 0. Thus, we continuously create and terminate actors, which process a total of 50,000,000 messages.

We also create one worker per ring that calculates the prime factors of 28,350,160,440,309,881, i.e., 329,545,133 and 86,028,157, to add numerical work load. It is worth noting that this operation is independent of any other actor and does not involve messages. The calculation requires about two seconds in our loop-based C++ implementation. Our tail-recursive Scala implementation of the prime factorization operates at the same speed, whereas Erlang needs almost seven seconds to finish the calculation.

In total, we create 20 rings. Each ring consists of 49 `chain_link` actors and one `master`. The `master` re-creates the terminated actors five times. Each `master` thus spawns a total of 245 actors. Additionally, there is one message collector and one worker per master. The message collector waits until it receives the result of 100 (20·5) prime factorizations and a *done* message from each master. Overall 4,921 actors are created, but no more than 1021 actors run concurrently.
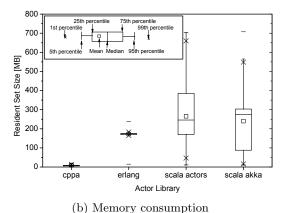
(a) Sending and processing time

(b) Memory consumption

Figure 3: Performance in a mixed scenario with additional work load

The following pseudo code illustrates the implemented algorithm.

```
master(Worker, Collector):
 5 times:
  Next = self
  49 times: Next = spawn(chain_link, Next)
  Next ! {token, 10000}
  Worker ! {calc, 28350160440309881}
  Done = false
  while not Done:
   receive:
    {token, X} =>
     if (X > 0): Next ! {token, X-1}
     else: Done = true
 Collector ! {master_done}

chain_link(Next):
 receive:
  {token, N} =>
   Next ! {token, N}
   if (N > 0) chain_link(Next)

worker(Collector):
 receive:
  {calc, X} =>
   Collector ! {result, fact(X)}
```

Figure 3(a) shows the runtime behaviour as a function of available CPU cores. An ideal characteristic would halve the runtime when doubling the number of cores. Erlang exhibits an almost linear speed-up, while `libcppa` and Akka decrease monotonically but non-linearly. Scala Actors increase in runtime after a minimum at eight cores, but accelerates again on twelve cores. Erlang performs well considering our previous observation that its prime factorization is more than three times slower. The efficient scheduling of Erlang, which is the only implementation in our measurement that performs *preemptive* scheduling, perfectly utilizes hardware concurrency up to ten core, when it reaches its global minimum. Akka is significantly faster than the standard library implementations of Scala but is slightly outperformed by `libcppa`.

Figure 3(b) shows the memory consumption during the mixed scenario. Both Erlang and `libcppa` have a very constant and thus predictable memory footprint. As it seems, Erlang's virtual machine pre-allocates about 180 MB and never needs to allocate additional memory during the run-

time. `libcppa` uses only a fraction of the memory compared to all other implementations and accounts for the benchmark's characteristics – constant number of actors and messages on average – as memory is released as soon as possible. Akka as well as Scala Actors exhibit a very unpredictable memory consumption, which cannot be explained by the benchmark characteristics, with an average memory consumption between 250 and 300 MB, and peaks above 600 MB.

8. LIMITATIONS INDUCED BY C++

Pattern matching has proven useful for a lot of different scenarios and is not limited to message handling in actor systems. We think our approach pushes the possibilities to emulate pattern matching using template metaprogramming to the limit. Yet, a library-based approach can only bring some of the power of pattern matching to C++ and is far less elegant than built-in facilities for instance known from Haskell or Scala. Furthermore, splitting the pattern part from the definition of the variables that holds matching parts makes our approach more verbose – and thus harder to read – than it could be with proper language support.

Due to the lack of pure functions in C++, we cannot check at compile time whether projection and conversion functions fulfill our requirement of not having side-effects and not throwing exceptions. Relying on conventions rather than enforcing requirements is a serious shortcoming and can lead to subtle bugs that are very difficult to find, even for expert programmers.

Furthermore, lambdas passed to `become` must not have references to the enclosing scope. Since `become` always returns immediately, all references are guaranteed to cause undefined behavior. Again, we have no chance of checking and enforcing this requirement, since the type of the lambda expression does not exhibit any information other than its signature. This problem is probably difficult to address in the language specification itself, but standardized attributes to annotate this kind of requirement would allow static analyzers to find related bugs with relative ease.

The adaption of language facilities previously found in functional programming languages like type inference, lambda expressions, and partial function application provided by `std::bind`, raised the expressive power of C++. In fact,

`libcppa` would most likely not be possible in the C++03 standard. In our opinion, the community should stay on this path of including more and more "functional" features, such as pattern matching and maybe monads, as they make asynchronous APIs very simple and potentially straightforward to implement. Although new features increase the overall size of the language, it allows developers to write cleaner and shorter code.

9. FUTURE WORK

One of the main strengths of the actor model is its abstraction over heterogenous environments. With the increasing importance of GPGPU programming – GPUs are known to outperform CPUs in computationally intensive applications –, it is a natural next step to provide facilities to create GPU compliant actors in `libcppa`. Since a GPU has its own memory, code and data are transferred to the GPU before executing an algorithm and the results must be read back after the computation is done. This management could be done by `libcppa`, since this workflow fits very well to the message passing paradigm. An GPU actor could define its behavior based on patterns, but would have to provide an additional GPGPU implementation, e.g., by using OpenCL. Such actors would be scheduled on the GPU rather than on the cooperatively used thread pool if an appropriate graphics card was found. Executing actors on a GPU would enable `libcppa` to address high-performance computation applications based on the actor model as well.

As for synchronous messaging, we want to relax the requirement of being forced to use timeouts. Instead, the runtime system could send an error message if the receiving actor has already exited.

Overall, we think `libcppa` can serve as a good tool for C++11 developers. As an additional future direction, we will focus on supporting ARM platforms and embedded systems by optimizing `libcppa` to be even more resource efficient. In this way, we hope to contribute our share to ease development of native, distributed applications in both high-end and low-end computing.

10. REFERENCES

[1] Agha, G. Actors: A Model Of Concurrent Computation In Distributed Systems. Tech. Rep. 844, MIT, Cambridge, MA, USA, 1986.

[2] Agha, G., Mason, I. A., Smith, S., and Talcott, C. Towards a Theory of Actor Computation. In *Proceedings of CONCUR* (Heidelberg, 1992), vol. 630 of *LNCS*, Springer-Verlag, pp. 565–579.

[3] Armstrong, J. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, KTH, Sweden, 2003.

[4] Bocchino, R. L., Adve, V. S., and Chamberlain, B. L. Software Transactional Memory for Large Scale Clusters. In *Proceedings of the 13th PPoPP* (New York, NY, USA, 2008), ACM, pp. 247–258.

[5] Coplien, J. O. Curiously Recurring Template Patterns. *C++ Report 7* (February 1995), 24–27.

[6] Doherty, S., Groves, L., Luchangco, V., and Moir, M. Formal Verification of a Practical Lock-Free Queue Algorithm. In *FORTE* (September 2004), vol. 3235, Springer, pp. 97–114.

[7] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Comput. 22*, 6 (Sept. 1996), 789–828.

[8] Haller, P., and Odersky, M. Event-Based Programming without Inversion of Control. In *Joint Modular Languages Conference* (2006), vol. 4228 of *LNCS*, Springer-Verlag Berlin, pp. 4–22.

[9] Haller, P., and Odersky, M. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theor. Comput. Sci. 410*, 2-3 (2009), 202–220.

[10] Halloway, S., and Bedra, A. *Programming Clojure*. Pragmatic Programmers, May 2012.

[11] Herlihy, M. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (Jan. 1991), 124–149.

[12] Herlihy, M., and Moss, J. E. B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA* (New York, NY, USA, May 1993), ACM, pp. 289–300.

[13] Hewitt, C., Bishop, P., and Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd IJCAI* (San Francisco, CA, USA, 1973), Morgan Kaufmann Publishers Inc., pp. 235–245.

[14] I.B.M. Corporation. IBM System/370 Extended Architecture, Principles of Operation. Tech. Rep. SA22-7085, IBM, 1983.

[15] ISO. ISO/IEC 14882:2011 Information technology — Programming languages — C++. Tech. rep., International Organization for Standardization, Geneva, Switzerland, February 2012.

[16] Meyers, S., and Alexandrescu, A. C++ and the Perils of Double-Checked Locking. *Dr. Dobb's Journal* (July 2004).

[17] Nyström, J. H., Trinder, P. W., and King, D. J. Evaluating Distributed Functional Languages for Telecommunications Software. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang* (New York, NY, USA, 2003), ERLANG '03, ACM, pp. 1–7.

[18] Rettig, M. Retlang. `code.google.com/p/retlang`, December 2010.

[19] Shavit, N., and Touitou, D. Software Transactional Memory. In *Proceedings of the fourteenth annual ACM symposium on PODC* (New York, NY, USA, 1995), ACM, pp. 204–213.

[20] Srinivasan, S., and Mycroft, A. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd ECOOP* (Berlin, Heidelberg, 2008), vol. 5142 of *LNCS*, Springer-Verlag, pp. 104–128.

[21] Torrellas, J., Lam, H. S., and Hennessy, J. L. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Trans. Comput. 43*, 6 (June 1994), 651–663.

[22] Typesafe Inc. Akka. `akka.io`, March 2012.

[23] Varela, C., and Agha, G. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Not. 36*, 12 (December 2001), 20–34.

[24] Veldhuizen, T. Expression Templates. *C++ Report 7* (1995), 26–31.