Singularity Design Note 5

Channel contracts

Governing message exchanges over channels

Channels are the fundamental way by which processes in Singularity communicate. A channel in Singularity consists of two distinct endpoints. Channels can be created dynamically and endpoints can be sent across channels themselves, allowing the dynamic creation of communication configurations. At any point in time, each channel endpoint is associated with a unique process. Channels are bidirectional conduits where sends are asynchronous, and receives can be blocking. Each channel is described by a static contract, governing the type of data and sequences of message exchanges between its endpoints. These contracts go far beyond what has been specified formally and checked automatically in the past. The main properties enforced by contracts are: 1) senders send messages expected by receivers, 2) receivers are ready to handle all messages allowed by the contract, 3) clients and servers that have been verified separately against the same contract C are guaranteed not to deadlock when allowed to communicate according to C.

1. Assumptions

This section states assumptions about channel semantics made implicitly in the channel contracts described here. These assumptions also influence the ability to statically verify that programs adhere to particular channel contracts.

1.1. Operational view of a channel

Each channel consists of exactly two endpoints. Endpoints are the only manifestation of channels seen by processes. Logically, a channel consists of two queues, one in each direction. Each endpoint uses one queue as its receive buffer. Sends through an endpoint are enqueued onto the receiving side's queue. Receives on an endpoint dequeue the first element in the corresponding receive queue.

1.2. Processes and endpoints

Each endpoint is associated with exactly one process at any given point in time. A process obtains channel endpoints via one of three means:

1. It creates a fresh channel, thereby obtaining both corresponding endpoints.
2. It receives the endpoint as a message through some channel.
3. It receives the endpoint at process startup as one of the initial arguments.

Correspondingly, a process can hand-off endpoints, either when starting a new process, or via sending the endpoint through a channel. Endpoints represent both an obligation and a capability for a process. The obligation imposed via an endpoint on a process is for the process to perform the message exchange that the channel contract specifies. The capability granted to the process is to

interact with the endpoint via sends and receives in any way allowed by the contract. Handing-off of endpoints corresponds to handing off both the obligation and capability to some other process.

1.3. Process termination

When a process dies, all channels for which the process holds an endpoint are closed. Closing a channel means marking the corresponding endpoints as closed. When a process tries to send or receive on a closed endpoint, the operation raises an exception that the process can decide to catch. Channel contracts do not explicitly mention channel closures. The contracts are written as if a channel remains open until there are no further message exchanges enabled.

1.4. Endpoint operations

1.4.1. Sends

A process can perform a send on an endpoint. The particular sends allowed on the endpoint at a given program point are restricted by the channel contract. All sends are asynchronous, thus a send operation completes immediately on the sending process.

1.4.2. Receives

A thread receives messages via an explicit **select receive** primitive. Select receives allow the thread to specify any number of channel endpoints and message combinations which it is ready to act upon. The select receive is blocking until a valid message combination is received, or one of the involved channels is closed, or some specified timeout occurs. If the select unblocks due to a closed channel, either an exception is raised, or a branch of the select receive explicitly handling closing conditions is executed.

1.4.3. Fairness of receives

We want to guarantee some fairness when multiple branches in a **select receive** are enabled. We thus assume that receives are fair in the following precise way:

A receive pattern case cannot be infinitely often enabled and not fire.

2. Channel contracts

A contract specifies a set of messages with associated parameter types and a set of states describing allowable sequences of message exchanges. Contracts are used to annotate channel endpoints. There are two sides to each contract, the importing view (client), and the exporting view (server). An importing endpoint for a contract C has type **imp**<C>, and the corresponding exporting endpoint has type **exp**<C>. In some contexts, it is necessary to describe endpoints whose contracts are statically unknown. Such dynamically typed endpoints have types **imp**<> and **exp**<>. These endpoints cannot be used until they are cast to a particular contract, at which case the contract is dynamically checked.

2.1. Example

Here is a contract for a name server:

```
contract NameServer {
    in message Register(String name, imp<NamedService> NameServiceChannel);
    in message Lookup(String name);

    out message RegisterAck;
    out message RegisterNack;
    out message LookupAck (imp<> serviceInstance);
    out message LookupNack;

    START: one {
      Register ?  -> REGISTER-ACK ;
```



```
    Lookup ? -> LOOKUP-ACK  ;
  }

  REGISTER-ACK: one {
    RegisterAck! -> START;
    RegisterNack! -> START;
  }

  LOOKUP-ACK: one {
    LookupAck !  -> START;
    LookupNack ! -> START;
  }
```

A contract is written in terms of messages and states. In each state, the contract specifies what messages may be sent or must be received by the server. A contract starts in the first listed state. For the NameServer contract this is the START state. In this state, a server implementing the NameServer contract must be ready to receive either a Register or a Lookup message. In the first case, the contract continues in the REGISTER-ACK state otherwise, the contract continues in the LOOKUP-ACK state. In either of these states, the server can choose to send either the corresponding Ack or Nack message. After that, the message exchange repeats, since the next states are again the start state. Message LookupAck returns a dynamically typed channel endpoint to the service that was found. The client would have to cast this endpoint to the contract it expects from this service, resulting in a dynamic check that the contract expected by the client does indeed match that of the service.

2.2. Contract syntax and semantics

Contracts are written from the viewpoint of the exporting party. Thus, if a contract states that it receives a message M at a certain point, then the exporting party must receive this message, and the importing party can send that message.

2.2.1. Basic contracts

A contract consists of message type declarations, as well as state declarations. Contracts are named and possibly generic over types (or other contracts). Syntactically, contracts have the following general form:

```
contract Id [BoundVars]  {
   MessageDeclarations

   StateDeclaration
}
```

Each message declaration specifies an optional direction, a message name and a sequence of message argument types. The direction is either **in** or **out**, limiting the use of the message to be either received or sent by the exporting side. Without a directional qualifier, the message can be used in both directions. The bound variables allow one to parameterize the message by types or contracts.

```
[in | out] message MessageId [BoundVars]( Type [Id], …, Type  [Id] );

BoundVars ::= < Id, …, Id >
```

A state declaration has the following general form:

```
StateId [BoundVars]: ( one | all ) {
  MessageSequence1
       …
```



```
   MessageSequence_n
}
```

The state identifier names the state. The state kind states whether the list of message sequences should be interpreted as a choice of **one** message sequence listed, or an arbitrary interleaving of **all** message sequences listed. In the case of a choice, message receives are interpreted as a choice made by the importer of the contract (the client), message sends are interpreted as a choice made by the exporter (the server).

If there are both receive and send choices present, we restrict the forms to two cases:

1. All receives are listed first, all sends are listed second.
   In this case, the choice of doing a send is with the importing side of the contract. The exporter must be willing to receive the listed receive messages, and can only send a message if the queue is either empty, or contains a message not listed among the receive messages.
   There is still a race in such a contract that needs to be resolved in the continuations of the message sequences starting with a send.

2. All sends are listed first, all receives are listed second.
   In this case, the choice of doing a send is with the exporting side of the contract. The importer must be willing to receive the listed send messages, and can only send a message if the queue is either empty, or contains a message not listed among the send messages.
   There is still a race in such a contract that needs to be resolved in the continuations of the message sequences starting with a receive.

In the case of **all** interleavings of messages sequences, the choice of order lies with senders of messages. For example, the snippet **all** { M! -> A; N! -> B; } would allow the sender to send message M and N in any order and the receiver has to be ready to receive them in any order. Similarly, if the interleaving specifies receives all { M?-> A; N? -> B; }, the choice of order lies with the sender. The receiver is willing to receive them in any order. For mixed interleavings of sends and receives, the same reasoning applies: the sender gets to choose the order.

Message sequences start with an action, followed by a continuation.

```
MessageSequence ::= Action Continuation
```

Actions are either sends **!** or receives **?** of messages declared in contracts. The send or receive symbols can be omitted if the message is declared with a direction. However, contracts may be easier to read if actions include the send/receive symbol on each message.

```
Action ::=
        MessageId [BoundVars]
    |   MessageId [BoundVars] !
    |   MessageId [BoundVars]?
```

Continuations are either empty (ending the message sequence) or consist of further actions, next states, or inline choices:

```
Continuation::=
        ;
    |   -> StateId [BoundVars] Continuation
    |   -> Action Continuation
    |   -> ( MessageSequence or … or MessageSequence )
```

The continuations in an inline choice must start with an action and all actions have to be of the same kind (all sends or all receives). Final states in a contract are those containing no message sequences.

2.2.2. Subroutine states

To reuse descriptions of finite message exchanges at several points in a contract, we allow a continuation to use a state as a sub routine. A state mentioned in a message sequence is a subroutine state, if it isn't followed by a semicolon. Here's an example contract using a subroutine state.

```
Start:  Process -> Start;

Process: one {
  M1? -> (A1 or A2)
  M2? -> (A3 or A4)
```

In the Start state, we first accept message sequences according to subroutine state "Process", then the protocol repeats to the Start state. The syntax of the Start state uses the short hand introduced in section 2.2.5.

A message sequence with a subroutine state evolves according to the named state. When all messages exchanges are exhausted according to that state, the contract evolves according to the remaining continuation. We require the named subroutine state to describe message exchanges of only finite length. Contracts not satisfying this restriction are considered ill-formed.

2.2.3. Joins

The semantics of subroutine states provide a way to express joins in the contracts. A join is like a barrier, waiting for a set of possible traces to end before resuming the continuation of the contract. If a subroutine state contains parallel message sequences, then the continuation after the subroutine state, starts only after those message sequences have all been exhausted.

2.2.4. Contract references

When referring to a contract C in types such as imp<C> or exp<C>, the contract reference C has the following form:

```
ContractRef ::=Id
        |  Id @ Id
```

The first form refers to a contract named *Id* in the implicit start state of that contract. The form *ContractId* @ *StateId* refers to the contract *ContractId* in state *StateId*.

2.2.5. Short hands

States consisting of a single message sequence can be written without kind and braces, and the message sequence doesn't have to start with an action.

```
StateId [BoundVars]:   ContinuationOnly
```

where

```
ContinuationOnly ::= ;
                   | StateId [BoundVars] Continuation
                   | Action Continuation
```

2.3. Contract verification

Writing asynchronous message passing code between clients and servers is difficult to get right. The purpose of specifying contracts in Singularity is to:

- Formally specify the interfaces between communicating parties. This serves both to document the interface for humans, as well as separating the two parties, allowing each one to be implemented independently against the contract.
- The contract is used as a formal specification to check that both the client and server code adhere to the contract. Adhering to the contract means sending messages with the appropriate arguments specified in the contract at the appropriate points in the interactions, as well as being ready to receive the possible messages at appropriate points. If two parties adhere to the contract, then any interaction

of the parties is guaranteed not to get stuck. Getting stuck formally means that a process would be waiting for a message that is never sent, or a process sending a message that is never received.

- The contract language and checking theory is carefully designed to allow modular reasoning. Modular reasoning is important in that it allows checking client and server code against contracts independently, and obtaining a guarantee that if the two are allowed to communicate, they won't get stuck. This ability is important, since it avoids having to check each combination of server and client code, which clearly is intractable.

We will use a combination of static and dynamic contract checking to obtain the benefits of writing such contracts in Singularity. Channel endpoints are statically typed in Singularity programs as either **imp**<C> or **exp**<C>. To verify a contract, we check that the program interacts with such endpoints according to the associated contract. Informally, the checker statically maintains the possible states in the protocols of each endpoint at each program point. At sends and receives, the checker then makes sure that the current states of the protocol allow the interaction.

Dynamic checking of contracts arises when we have untyped endpoints **imp**<>. Before using such an endpoint, a program needs to cast it to a particular contract **imp**<C>, analogously to downcasting from type Object to some particular class B. In order to implement such a contract check dynamically, it will be necessary to associate a runtime representation of the contract with a dynamically typed endpoint **imp**<>. The dynamic check then enforces that the two contracts have the same type of messages and message arguments, as well as conforming message sequences.

3. Examples

3.1. Traditional RPC style interface

A traditional RPC style interface accepts one of k messages (corresponding to the different methods), and provides a single reply to each before the contract repeats. Such contracts have the following generic form:

```
contract GenericRPC {
    in message M1(…);  // method M1
    out message A1(…);  // results of call to M1

    …
    in message Mk(…);  // method Mk
    out message Ak(…);  // results of call to Mk


    Start:   SingleRequest -> Start;

    SingleRequest: one {
            M1? -> A1!
            M2? -> A2!
            …
            Mk? -> Ak!
    }
```

In state Start, the message exchange evolves according to subroutine state SingleRequest, and then it repeats. State SingleRequest accepts one of the k request methods and replies with the fixed corresponding answer (corresponding to the RPC's return).

This contract is simple in that it accepts all orders of message requests and responds in a fixed way. Contracts can be much richer in that the acceptable messages and responses can depend on the history of previous interactions. One particular refinement that is useful is to make different possible responses explicit in the contract as follows:


```
SingleRequest: one {
        M1? -> (A11! or A12! or A13! … )
        M2? -> (A21! or A22! or A23! …)
        …
        Mk? -> (Ak1! or Ak2! or Ak3! …)
}
```

3.2. Travel reservation

This example consists of a ReservationSession contract that is a finite exchange of messages between a requestor and an agent, as well as a contract to establish such a session, the RequestSession contract. The message payloads are not detailed in the ReservationSession contract.

```
contract ReservationSession {
    in message Request(DateRange, …);
    in message Cancel;
    in message Confirm;
    out message Succeed(ReservationId, …);
    out message Fail;
    out message Cancelled;

    START:   Request? -> DECIDE;

    DECIDE: one {
            Succeed! -> SUCCESS;
            Failed! -> SINK;
            Cancel? -> Cancelled! -> END;
    }

    SUCCESS: one {
            Confirm? -> END;
            Cancel? -> END;
    }

    SINK -> Cancel? -> END;
}
```

An agent implementing a ReservationSession starts out by receiving a Request message. It then decides to either send a Succeed or a Failed message, or it can decide to receive a Cancel message at this time if such a message is sent by the client. After having sent Succeed, the agent awaits a confirmation or cancel message from the client. In the case of a Failed message, a Cancel message is also awaited, confirming the failure. Without other support to drop messages, this is necessary here, because a Cancel and a Failed message could cross. Thus the agent has to receive it at this point. Here we show some pseudo code for how a client could interact with a ReservationSession.

```
// client example code

void Reserve( imp<ReservationSession> session ) {

    session.Request(…);
```



```
    select receive (session) {
            case Succeed(rid, …):
                    if (…) {  session.Confirm(…); } else { session.Cancel(..);
                    break;
            case Failed:
                    session.Cancel(…);
                    break;
            timeout(2secs):
                    // do some stuff
                    session.Cancel(…);
                    // at this point, we still need to finish the interaction, but we are left
                    // only with receiving these messages. We probably need some
                    // functionality to "flush" a set of unreceived messages if the receiver
                    // wants to ignore them.
                    select receive (session) {
                            case Succeed: break;
                            case Failed: break;
                            case Cancelled: break;
                    }
}
```

Below is the contract for obtaining new sessions. This contract provides new ReservationSession endpoints to the requestor, if a session is available.

```
contract RequestSession {
    in message RequestSession;
    out message SessionEndpoint( imp<ReservationSession> );
    out message Unavailable;

    START:   RequestSession? -> DECIDE;

    DECIDE: one {
            SessionEndpoint! -> START;
            Unavailable! -> START;
    }

}
```

3.3. Alternative NameServer 1

The NameServer contract in Section 2.1 returns a dynamically typed endpoint. The dynamic check to see if the contract of the returned endpoint matches the client's need is done in the client. Below, we sketch an alternative formulation of the NameServer contract that pushes the conformance check to the NamedService provider. Here we make use of messages and states that are parameterized by contracts. In this formulation, the client sends a Lookup message to the NameServer containing a runtime representation of a contract k. The NameServer is written to be generic in the actual contract type C. The client can thus instantiate the Lookup message to a particular contract (say K), provided he sends a runtime representation of contract K in the Lookup message. The client will then receive a typed endpoint of type **imp**<K> in the LookupAck message. In this scenario, the client does not need to perform a runtime contract check. The contract check is done by the NamedService provider that provides a new endpoint to the NameServer.

```
contract NameServer {
    in message Register(String name, imp<NamedService> NameServiceChannel);
```



```
    in message Lookup<C>(String name, Contract<C> k);

    out message RegisterAck;
    out message RegisterNack;
    out message LookupAck <C>(imp<C> c);
    out message LookupNack;

    START: one {
       Register ?  -> REGISTER-ACK ;
       Lookup<C>? -> LOOKUP-ACK<C> ;
    }

    REGISTER-ACK: one {
       RegisterAck! -> START;
       RegisterNack! -> START;
    }

    LOOKUP-ACK<C>: one {
       LookupAck<C>!  -> START;
       LookupNack! -> START;
}
```

```
contract NamedService {
    in message GetEndPoint<C>(Contract<C> k);

    out message NewEndPoint<C>(imp<C> c);
    out message Refused;

    START: one {
       GetEndPoint<C> ?  -> DECIDE<C> ;
    }

    DECIDE<C>: one {
       NewEndPoint<C>!  -> START;
       Refused! -> START;
}
```

3.4. Alternative NameServer 2

So far we had name servers where the contract check is performed by the client or by the service provider. This third formulation places the contract check at the NameServer itself. In this formulation, we are using generic NamedService contracts. Here, the entire NamedService contract is parameterized by the contract C of the endpoint provided by the service, rather than leaving the parameterization to each interaction as in Section 3.3.

```
contract NamedService<C> {
    in message GetEndPoint ();

    out message NewEndPoint (imp<C> c);
    out message Refused;

    START: one {
       GetEndPoint ?  -> DECIDE ;
    }

    DECIDE: one {
```



```
        NewEndPoint !  -> START;
        Refused ! -> START;
}
```

The NameServer contract looks a bit different as well. Registering the service now involves passing a contract representation to the name server. The nameserver stores that contract representation along with the NamedService endpoint in order to check the contract provided by a client request against the one provided by the named server.

```
contract NameServer {
    in message Register<C>(String name, imp<NamedService<C> > NameServiceChannel,
                           Contract<C> k);
    in message Lookup<C>(String name, Contract<C> k);

    out message RegisterAck;
    out message RegisterNack;
    out message LookupAck <C>(imp<C> c);
    out message LookupNack;

    START: one {
        Register<C> ?  -> REGISTER-ACK ;
        Lookup<C>? -> LOOKUP-ACK<C> ;
    }

    REGISTER-ACK: one {
        RegisterAck! -> START;
        RegisterNack! -> START;
    }

    LOOKUP-ACK<C>: one {
        LookupAck<C>!  -> START;
        LookupNack! -> START;
}
```

3.5. Return channel example

Even though at any point in time, a single thread interacts with a particular channel endpoint, there are nevertheless scenarios where multiple threads indirectly share a channel endpoint (via mutual exclusion). To make the window of mutual exclusion as small as possible, it is useful to separate out longer conversations using explicit reply channels. Here's the contract for a service accepting a number of requests. The followup conversation (if any) happens on other channels passed during the request. This allows a thread to obtain the endpoint, send a request, release the endpoint to another thread, while continuing the conversation on the reply channel.

```
contract SharedServer {
    in message Request1(imp<Reply1> replyChan, …);
    in message Request2(imp<Reply2> replyChan, …);
    in message RequestWithoutReply(…);
    …

    START: one {
        Request1 ?  -> START ;
        Request2 ? -> START ;
```



```
    RequestWithoutReply ? -> START ;
    …
}
```

To make the window of mutual exclusion as small as possible, this service only has *in* messages, since all replies are sent along session channels passed in the request message if necessary. The reply channel contracts Reply1, Reply2, … can be arbitrarily complicated, from a single message, to an arbitrary exchange. Note that since the reply channel is now dedicated, there is no need for further reply channels on such exchanges. Here's an example Reply contract: Note that the reply contract is written in terms of the thread receiving the reply.

```
contract Reply1 {
    in message Answer (…);
    in message Clarify( …);
    in message UltimateAnswer(…);

    out message Case1(…);
    out message Case2(…);


    START: one {
        Answer ?  -> END ;
        Clarify ? -> CLARIFYREQUEST ;
    }

    CLARIFYREQUEST: one {
        Case1 ! -> UltimateAnswer ? -> END ;
        Case2 ! -> UltimateAnswer ? -> END ;
    }
```

The Reply1 contract states that the server will reply with either an Answer message, ending the reply exchange. Alternatively, the server could send a Clarify message to which the requester answers with either message Case1 or Case2, to which the server then replies with UltimateAnswer. Note that there are no further reply channels are needed for these exchanges.

3.6. A streaming protocol

This protocol is a streaming protocol from the importing side to the exporting side. The protocol contains explicit flow control that allows the exporting side to block the sender with a Xoff, and later unblock him with a Xon message.

```
contract StreamXonXoff  {
    in message Request (…);
    out message Xoff( …);
    out message Xon( …);

    in message XoffAck();


    START: one {
        Xoff ! -> WAIT;
        Request ? -> START;
```



```
    }

    WAIT:  XoffAck? -> Xon ! -> START;
```

The mixed choice in the START state is a choice by the exporting side (since the sends are listed first). The exporting side can thus be implemented as follows:

```
exp<StreamXonXoff> e;

for (;; ) {
  if (e.QueueSize > k) {   // decide whether to send or receive

    e.Xoff(); goto Catchup;  // send Xoff, goto Catchup

  }

  receive Request(…) from e; Work(…);   // handle the request

  continue;

Catchup:

   select receive {

      case: Request(…): Work(…); goto Catchup;;

      case: XoffAck():
         e.Xon();
         continue;
   }
```

The code decides whether to receive a request or send a Xoff message depending on the queue size. If a Xoff is sent, we enter a loop where we are willing to receive a Request until we receive the XoffAck message which specifies that there are no more request left in the queue.

4. Programming language support of contracts

4.1. Method signatures

[To describe how method signatures modularly describe either the pre/post state of endpoints, or a summary of the actions performed on the protocol.]

4.2. Object invariants

[To describe the necessary invariants on contract state for endpoints held in objects.]

4.3. Endpoint abstractions

[To describe how the channel endpoint interactions can be extended to arbitrary objects, thereby enabling abstraction over concrete endpoints.]

