Gneiss
A nice component framework in SPARK

Johannes Kliemann
FOSDEM, Brussels, 2020-02-02
Component-based Architectures

**Trusted Components**

- Can’t reimplement everything
- **Solution: software reuse**
  - Untrusted software (gray)
  - Policy object (green)
  - Client software (orange)
- **Policy and proxy components**
  - Formally verified
  - Limited complexity
Ensuring Correctness

Prerequisites

- **Correctness by proof**
  - Absence of runtime errors
  - Functional correctness

- **Tools**
  - Formalization language
  - Mapping between implementation and proof

- **Reusability**
  - Proofs require effort
  - Abstraction from actual platform

- **Provability**
  - Formal specification
  - Manageable complexity
  - Deterministic behaviour
Correctness by Proof and Tools

**SPARK**

- **Programming Language**
  - Based on Ada
  - Compilable with GCC and LLVM
  - Customizable runtime
  - Contracts (preconditions, postconditions, invariants)

- **Verification Toolset**
  - Absence of runtime errors
  - Functional correctness

```plaintext
function Abs (I : Integer)
return Integer
with
  Pre => I > Integer'First,
  Post => Abs'Result >= 0;

procedure Inc
  (I : in out Integer)
with
  Pre    => I < Integer'Last,
  Post   => I = I'Old + 1,
  Global => null;
```
Provability and Reusability

Gneiss

■ Reusability
  ▪ Platform abstraction
  ▪ Interface mappable to multiple different semantics
  ▪ Only dependencies satisfiable by all platforms

■ Provability
  ▪ Platform formalization
  ▪ Assumptions coarse enough to be valid on multiple platforms
  ▪ Assumptions strong enough to ease proving
Example: Block Client
Block Devices
Client Interface

- **Block device**
  - Storage device of equally sized blocks
  - Block size is typically 512 or 4096 bytes

- **Packet descriptor**
  - Starting block number
  - Amount of blocks
  - Read/Write/Sync/Trim
  - Memory location

- Create packet descriptor
- Allocate memory for request
- (write data)
- Send request to block device
- Receive answer from block device
- (read data)
Gneiss Block Client
Formalizing properties

- Formalize properties of platform API
  - Packet object is needed
  - Packet object can always be initialized
  - Request memory must be allocated separately
  - Memory allocation might fail
  - Submitting must be checked
  - Submitting works always if ready

```java
packet = Packet_descriptor(
    WRITE, start, count);
try {
    packet.alloc_packet(
        block_size * count);
    if(ready_to_submit()){
        submit(packet);
    }
} catch (Alloc_Error) { }
```
Gneiss Block Client
**Formalizing properties**

- **Define packet type**
  - No exceptions, allocation success is a property

- **Define precondition from formalized properties**
  - Packet must be allocated
  - And the platform must be ready

```pascal
type Packet is record
  Start     : Natural;
  Length    : Positive;
  Op        : Operation;
  Allocated : Boolean;
end record;

function Ready return Boolean;

procedure Submit (P : Packet) with
  Pre => P.Allocated 
  and then Ready;
```
Packet properties can be changed by the programmer

- Allocation status can be set without actually successfully allocating
- Packet can be submitted multiple times

Submit does not change the platform state

- Calling `Submit` should invalidate `Ready`

```pascal
P := Packet'(Start => 0,
             Length => 1,
             Op => READ,
             Allocated => True);
if P.Allocated and then Ready
then
  Submit (P);
  Submit (P);
end if;
```
Gneiss Block Client
Formalizing properties

- Use state enum instead of boolean
- Encapsulate Packet type
  - Can only be changed by platform calls
  - Can only be created in state Empty
  - Cannot be copied (limited)

```plaintext
type Packet is limited private;
type Packet_State is (Empty, Allocated);
function Create
  (Start  : Natural;
   Length: Positive;
   Op     : Operation)
  return Packet
with
  Post => State (Create'Result) = Empty;
function State (P : Packet)
  return Packet_State;
```
Gneiss Block Client
Formalizing properties

- Submit changes packet state
- Submit changes platform state
  - **Ready** depends on platform state
  - Once Submit is called, **Ready** must be checked again

function Ready return Boolean with
  Global => (Input => Platform);

procedure Submit
  (P : in out Packet) with
  Pre => State (P) = Allocated
  and then Ready,
  Post => State (P) = Empty,
  Global => (In_Out => Platform);
■ write might fail
  ▪ ENOSYS (not implemented)
  ▪ EINVAL (wrong argument)
  ▪ EFBIG (offset out of file)
  ▪ EBADF (bad file descriptor)
  ▪ EAGAIN (out of resources)

■ No way to make sure it succeeds, submit must be able to fail, too

struct block_packet packet = {
  0, 1, WRITE, 0};
int result;
packet.ptr = malloc (block_size * packet.len);
if (packet.ptr){
  result = write(fd, &packet);
}
Submit must be able to fail

- It might change the packet state or leave it as is
- An unsuccessfully submitted packet can be submitted again

```procedure
Submit
(P : in out Packet)
with
Pre    =>
        State (P) = Allocated,
Post   =>
        State (P) in
        Empty | Allocated,
Global => (In_Out => Platform);
```
Gneiss Block Client
Adapting the first platform

- Both platforms have different semantics
- The second platform cannot be expressed with the first one
- But the first one can be expressed with the second one

```plaintext
procedure Submit
  (P : in out Packet)
is
begin
  if Ready then
    Submit_Native (P);
  end if;
end Submit;
```
Gneiss
Summary

- Asynchronous, event based
- Supports capabilities
- Callbacks via generics
- Limited dynamic resource allocation
  - Platform dependent
- No memory pressure
- No aliasing

- Multiple platforms
  - Genode
  - Linux
  - Muen

- Interfaces
  - Log client/server
  - Block client/server
  - Timer client
  - Message client/server
  - Shared memory
Questions?

Johannes Kliemann
kliemann@componolit.com

@Componolit · componolit.com · github.com/Componolit