### IMPLEMENTING TRUE SEPARATE COMPILATION THE BLECH MODULE SYSTEM

#### FRIEDRICH GRETZ FRANZ-JOSEF GROSCH

SYNCHRON, NOVEMBER 2020 - ONLINE-

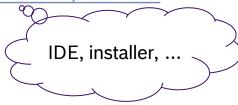


### The Blech synchronous language Blech Update since Synchron '19

- ► Visit us online: <u>www.blech-lang.org</u>
- Check out the implementation at <u>https://github.com/boschresearch/blech</u>
  - Compiles on all platforms! (Linux, Mac, Windows)
  - ► A VS Code plug-in can be found at <u>https://github.com/boschresearch/blech-tools/releases</u>

blog articles, documentation

- ► Follow us at <a href="https://twitter.com/BlechLanguage">https://twitter.com/BlechLanguage</a>
- ► Get in touch with us by email or <u>https://blech-lang.slack.com</u>



Blech  $\rightarrow$  C compiler



# Talking about separate compilation in Blech

#### Blech, imperative synchronous programming!

Friedrich Gretz Robert Bosch GmbH Corporate Research Renningen, Germany Friedrich.Gretz@de.bosch.com Franz-Josef Grosch Robert Bosch GmbH Corporate Research Renningen, Germany Franz-Josef.Grosch@de.bosch.com

Forum on specification & Design Languages 2018

- introduction of black-box activities,
- causality only based on input/output interface

FORUM ON SPECIFICATION AND DESIGN LANGUAGES - FDL 2020

#### Synchronized Shared Memory and Procedural Abstraction: Towards a Formal Semantics of Blech

F. Gretz and F-J. Grosch (Bosch Corporate Research) and M. Mendler and S. Scheele (Bamberg University)

Forum on specification & Design Languages 2020

 provide a formal semantics for this kind of procedural abstraction





# Talking about separate compilation in Blech

#### ► Today:

- ► a (synchronous) program is not just one file collecting all activities
- ► software architecture, separation of concerns, reuse of "packages" or "libraries" of software
- → modules
- Organising code in files and collections thereof is nothing new: e.g. Java classes + JAR
- Engineering task:
  - what granularity of name spaces and access rights do we need?
  - ► how does this integrate with C?
  - how does this integrate with a synchronous language and causality checking?



#### Talking about separate compilation in Blech no external dependency Running example: RingBuffer management, no local imports, upstream module ( no shadowing import rb "data structures/ringbuffer" export to downstream client-code module exposes SlidingAverage simple visibility param Threshold: nat32 = 10000 properties gathered in one activity SlidingAverage (value: nat32) (average: nat32) declaration at the var buf: rb.RingBuffer = rb.initialise() beginning repeat if value <= Threshold then rb.push(value)(buf) access through given name rb end average = rb.average(buf) await true end

end



## Module system design challenges 1. Mapping names

Blech – hierarchical name spaces	
data_structures	void blc_data_
ringbuffer.blc	blc_nat32 val
push (value: nat32) (buf: RingBuffer)	struct blc_da
	)
	blc_slidingaver
slidingaverage.blc	blc_nat32 val
rb	blc_nat32 * a
SlidingAverage (value: nat32) (average: nat32)	)

#### C – flat global name space

structures\_ringbuffer\_push ( lue, ta\_structures\_ringbuffer\_RingBuffer \* average

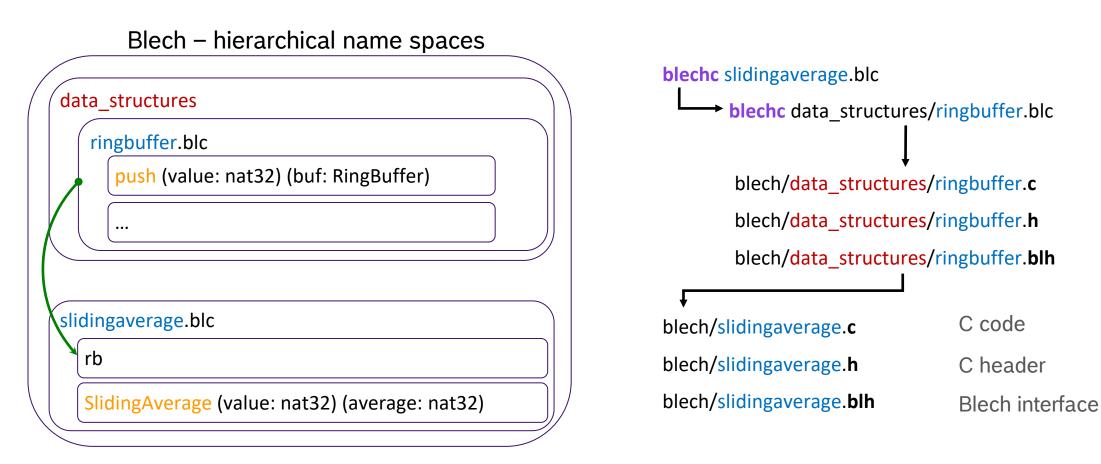
rage\_SlidingAverage

lue,

average



## Module system design challenges 2. Compiling dependencies automatically





Blech sources available

blechc slidingaverage.blc

blech/slidingaverage.h

#### blech/slidingaverage.blh

Blech header + C source

**blechc** slidingaverage.blc blechc blech/ data structures/ ringbuffer.**blh** blech/slidingaverage.c blech/slidingaverage.h blech/slidingaverage.blh blech/data structures/ringbuffer.c blech/data structures/ringbuffer.h Blech header + C object

**blechc** slidingaverage.blc blechc blech/ data structures/ ringbuffer.**blh** blech/slidingaverage.c blech/slidingaverage.h blech/slidingaverage.blh blech/data structures/ringbuffer.o blech/data structures/ringbuffer.h



Blech sources available

all sources available to the programmer

Blech header + C source

- API of imported Blech module is available
- Blech implementation is secret
- generated C code is available

Blech header + C object

- API of imported Blech module is available
- Blech implementation is secret
- ► C code is secret



#### Implementation: ringbuffer.blc

module exposes initialise, push, average

const Size: nat8 = 10

struct RingBuffer var buffer: [Size]nat32 var nextIndex: nat8 var count: nat8 end

function initialise () returns RingBuffer
 return { }
end

```
function push (value: nat32) (rb: RingBuffer)
  rb.buffer[rb.nextIndex] = value
  rb.nextIndex = rb.nextIndex + 1
  if rb.count = Size then // ringbuffer ist completely filled
    rb.nextIndex = rb.nextIndex % Size
  else
    rb.count = rb.count + 1
  end
end
```

```
function average (rb: RingBuffer) returns nat32
var idx: nat8 = 0
var avg: nat32 = 0
while idx < rb.count do
    avg = avg + rb.buffer[idx]
end
return avg / rb.count
end</pre>
```



Interface: ringbuffer.blh (generated by blechc)

signature

type RingBuffer the type is used by an exposed function and therefore is *implicitly* exposed function initialise () returns RingBuffer function push (value: nat32) (rb: RingBuffer) functions were *explicitly* exposed function average (rb: RingBuffer) returns nat32

the module constant "Size" is not exposed at all and unknown outside the ringbuffer module



#### What to do with singletons?

module exposes Monitor

```
@[CFunction (binding = "wifi_is_online()", header = "wifi.h")]
extern singleton function wifilsOnline () returns bool
```

```
activity Monitor () (leds: LEDs)

repeat

leds.wifiLed = wifilsOnline()

await true

end

end
```

#### signature

type LEDs

singleton wifilsOnline

singleton wifilsOnline activity Monitor (leds: LEDs)

Monitor must not be called concurrently with anything that uses wifilsOnline (including itself).



# Module system design challenges White-box unit testing

internal import rb "ringbuffer"

@[EntryPoint]
activity TestPush ()
var buf: rb.RingBuffer = rb.initialise()
var i: nat8 = 0
repeat
assert buf.nextIndex < rb.Size
assert buf.nextIndex == i % rb.Size
assert buf.count >= 0
assert buf.count <= rb.Size</pre>

```
rb.push(42)(buf) // the value is irrelevant
i = i + 1
```

```
if i < rb.Size then assert buf.count == i
    else assert buf.count == rb.Size end
    await true
    until i == 255 end
end</pre>
```

#### Internal import

- only possible if Blech code is available
- allows to separate testing code from product code



### Design pragmatics Layered Architecture

#### data\_structures

ringbuffer.blc

slidingaverage.blc

#### main.blc

import sa "slidingaverage"

#### @[EntryPoint]

activity Main (sensor: nat32) (sensorAverage: nat32) run sa.SlidingAverage(sensor)(sensorAverage) end

- absence of import cycles checked
   automatically
- differentiate modules and programs
  - programs contain an entry point and cannot be imported, no blh is generated
- separate testability of each module (or layer)



### Summary

- Module = file
- Everything visible within file (or within internal import)
- ► A declaration is either exposed or not (opaque types / singletons automatically exposed if necessary)
- ► Generation of a "Blech API" (\*.blh) which
  - does not reveal implementation details
  - suffices for downstream code generation
- Layered architecture
- Modules could be wrapped to packages (cf. <u>Blog</u>)
- $\blacktriangleright$  Modules organise name spaces but they do not address generic data structures  $\rightarrow$  future work
- ► Implementation <u>work in progress</u>, planned release end of 2020.



### Design pragmatics Syntax

- dependencies clearly visible in code instead of external project configuration files
- simple visibility properties
- gathered in one declaration at the beginning



### Design pragmatics White-box testing



#### ► recap:

- ► FDL'18 explained the principle of black-box activities, causality only based on input/output interface
- ► FDL'20 provide a formal semantics for this kind of procedural abstraction

#### ► Today:

- ► a (synchronous) program is not just one file collecting all activities
- ► software architecture, separation of concerns, reuse of "packages" or "libraries" of software
- ► → collect types, activities, functions into a module (= file)
- collect modules into library
- ▶ this is common place for standard languages (Java JARs, Rust crates, ...)
- our main challenges:
  - mapping to C where all names are global without producing clashes
  - compiling everything that a module/program needs automatically, unlike C/C++ where you need to specify all dependencies in a makefile manually
  - lift black-box approach to modules, meaning modules may be precompiled and the compiler relies only on the module's interface which we call "Signature"

© Robert Bosch GmbH 2020. All rights reserved, also regarding any disposal, exploitation, reproduction, editing, distribution, as well as in the event of applications for industrial property rights



- ► Design choices in detail
  - ▶ name mangling: encode folder structure, module name into the name of every static element
  - exposing (types) explicitly vs implicitly vs not-at-all
    - within module scope (file scope) everything is visible
    - functions, activities, constants are either exposed to the client or not
    - types which are not explicitly exposed but are required by the parameter list of exposed functions/activites are exported as abstract types (i.e. just names)
    - for unit testing, special white-box import to keep implementation file and test file separate (source code needed, signature file insufficient, but that is given for testing)
  - ► singletons
    - activities which access global (external) memory become singleton in Blech, activities calling singletons become singletons themselves
    - in order to causality check such singletons (prevent concurrent use to itself) the signature must contain the "reason" i.e.
       names for why they are singleton



#### Lessons learnt? or why are we telling you this??

- engineering challenge
- causality analysis with module signatures, no global analysis required
- separate compilation with precompiled sources for synchronous lang
- make sure non-exported elements do not leak to the outside
- ► KISS
- Status of implementation?
- ► Future work: generics (orthogonal to modules)

