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

***Abstract*—We discuss how separate compilation and usage of structured data can be achieved in a synchronous imperative programming language for embedded systems. Our focus is on the practical aspects of these questions regarding expressiveness in programs as well as causality analysis and code generation. The approach is illustrated by means of examples written in our new language "*Blech*".**

***Index Terms*—embedded software, synchronous programming languages, separate compilation, causality analysis, structured data**

I. Introduction

Synchronous languages have in certain cases been successfully used for embedded software programming in industrial projects in the past [4] but they remain an exotic tool known only to experts. They are not part of a general embedded programming curriculum nor are they used outside special safety-critical embedded applications in industry. This is particularly regrettable because the key features of these languages, such as reactive, concurrent programming and the guarantee of causality, could mean a conceptual leap forward in programming of most of today's embedded applications. Esterel [8] mostly targeted hardware design, not application-level software development. As of today its development has ceased and no up-to-date compiler is available. Quartz [10] can be regarded as a successor of Esterel but has the same hardware focus, limiting its applicability in software development. Scade [6] is a an industrial grade tool but it is tailored to safety critical applications in particular industrial domains. Its dataflow orientation caters to its special target audience but may not be the best choice for many other embedded software programmers. Céu [9], a recently developed imperative language, aims at the programming of reactive embedded applications but it provides no causality guarantees. We therefore propose a new, imperative, synchronous, and purely software-development oriented programming language called *Blech*[1]. From our point of view, the main benefit of synchronous programming is the automatic causally correct composition of concurrent subprograms. At the same time, this is the single most challenging feature to accommodate for when designing the semantics and implementing compilers for such languages.

[1]Blech is German and colloquially translates to "bare metal" highlighting our focus on deeply embedded architectures.

978-1-5386-6418-6/18/$31.00 ©2018 IEEE

In the next section we familiarise the reader with our language *Blech*. After that, we turn our focus on some key aspects regarding causality. We discuss prior work in Section III. Subsequent sections describe our design choices and their consequences.

II. Blech

The following listing shows a sample program written in *Blech*. It simply runs a PID controller and toggles between two modes of operation depending on user input. Despite its modest functionality, this program demonstrates different language features that may give an impression of what can be expressed in *Blech*. Below we explain the relevant details.

```
enum Button
  Left default
  Right
end

struct PID
  var KI: float32 = 0.0
  var KD: float32 = 0.0
  var KP: float32 = 1.0
  var dt: float32 = 1.0
with
  function this:calc
           ( pv: float32, sp: float32 )
           ( priErr: float32,
             intl: float32, cmd: int32 )
    let error = sp - pv
    let der = (error - priErr) / this.dt
    intl = intl + (error * this.dt)
    cmd = this.KP * error + this.KI * intl
          + this.KD * der
    priErr = error
  end

  activity this:control(pv:float32,sp:float32)
                       (cmd:int32)
    var priErr: float32 = 0
    var intl: float32 = 0
    repeat
      this:calc(pc, sp)(priErr, intl, cmd)
      await true
    end
  end
end

@[EntryPoint]
activity main ()()
  extern let sensor: float32
```

```
  extern var output: float32
  extern let pressed: event(Button)
  var pid: PID
  let sp: float32 = 42.17                                    41

  repeat
    cobegin
      await let b = pressed, b == .Left
    with weak                                                46
      pid.KP = 0.9; pid.KI = 0.0
      run pid:control(sensor, sp)(output)
    end

    cobegin                                                  51
      await let b = pressed, b == .Right
    with weak
      pid.KP = 0.5; pid.KI = 0.5
      run pid:control(sensor, sp)(output)
    end                                                      56
  end
end
```

Reactions: A *Blech* program is triggered by the runtime environment at every tick of some external clock. The program responds to a tick by performing a *reaction step* which usually will read input data and compute output data from it. Following the synchrony assumption, the runtime environment must ensure that all input variables are sampled at the beginning of the reaction and remain unchanged while the reaction executes. The reaction starts in the entry point activity *main* in line 36. Activities are a kind of subprograms in *Blech*. We also use functions and will explain the difference later on. An activity will usually declare some local variables.

Declarations: Declarations are indicated by the `var` and `let` keywords. The former means that the variable is indeed mutable whereas the latter declares immutable (or read-only) data[2]. Additionally, `extern` tells the compiler that the variable is defined outside this program and we can simply expect it to be in our scope after linking. This is necessary whenever we receive data from the runtime environment or write data to it as is the case with sensor readings and controller commands. Every declaration must either provide a data type explicitly (as in lines 37 – 41) or it may be deduced if the right-hand side of the declaration uniquely determines the data type. Also every declaration will automatically initialise a variable to the type's default value. This is 0 for numerical types and in case of the *PID* structure the default value is given in the type declaration (lines 7 – 10).

Events: A special, built-in generic data type is the `event` type. Events are a special form of an optional data type wherein the event either is absent or it is present and carries some payload. In order to become present during a reaction, an event must be *emitted* either by the runtime environment if it is an external event, or by using a special `emit` keyword for internal events. In any case, the runtime clears all events at the end of a reaction so that an event does not persist from one reaction to another unless it is re-emitted again at the beginning of the next reaction. In line 39 we define an external event which indicates that a user has pressed a button. The payload is of type *Button*, defined by the enumeration in lines 1 – 4, and tells us which of the two buttons has been pressed. The use of events will be explained in the next paragraph.

Statements: The *main* activity consists of an infinite `repeat..end` loop. In the body of that loop, the control flow is forked into two branches using the `cobegin..with..end` statement. Conceptually, i.e. from the programmer's point of view, the two branches are executed concurrently. The compiler, however, will sequentialise these branches into one sequence of instructions after an automated causality analysis, which ensures that such a sequentialisation exists. Generally, the branches of a `cobegin` statement will join when each of them has terminated. The `weak` keyword may be used to indicate branches which may be aborted. Let us see how this works out in lines 44 – 49. In line 45, the branch consists solely of an `await` statement. When control flow reaches an `await` statement, the execution along that branch is stopped. A reaction is finished when every branch has hit an `await` statement. Upon the next tick, a new reaction starts and all branches resume execution from the `await` statements that they have ended in previously. Every `await` statement is equipped with some condition. Often this condition is simply `true` which means that the program just waits for a new trigger to continue (e.g. in line 30). The condition in line 45 is more complex. Remember that *pressed* is an event which may or may not be present, so the first part checks for presence and in such case copies the payload to the local immutable variable *b*. Then we check whether the payload's value is *Button.Left*[3]. If both the presence test and the comparison succeed, the execution continues beyond the `await` statement and in this case the branch terminates. If either fails, the control flow remains at the `await` statement and waits for the next reaction to start. Concurrently, in lines 47 – 48, some fields of the *pid* struct are re-initialised and then the *control* activity is started that operates on *pid* and some given arguments. The effect of making the `with` branch `weak` is that in the reaction where the user has pressed the left button, the concurrently running *control* activity is executed until it finishes its current reaction step and then completely terminated, the branches are joined and control flow proceeds to the next statement in line 51. Essentially, this is a simple way to guard a repetitive, possibly infinite, behaviour with some abortion condition. Note that the same behaviour could have been expressed using a (non-immediate) `abort` statement. Regrettably, a discussion of the various notions of synchronous preemptions, cf. [10, p. 34], is beyond the scope of this paper. In general, `cobegin` allows any number of `with` blocks and any block can be made `weak`. Summing up, the *main* activity defines two modes of operation given by the setting of the PID controller in lines 47 and 54 and toggles between these states whenever the user presses the right or the left button.

[2]Here `let` should not be confused with the same keyword in functional languages where it binds a free variable in a subsequent subexpression.

[3]By default enumerations are opened and hence their tags may be accessed without specifying the type name.

Subprograms: Let us look in more detail at the different kinds of subprograms in *Blech*, namely functions and activities. Generally, subprograms will have a list of read-only parameters (inputs) and a list of read-write parameters (outputs). When calling a subprogram, all arguments are passed by reference. The difference is that functions are *instantaneous*. This means that a function must run to completion during a reaction step. It may not use `await` statements or call activities. Functions in *Blech* will usually encapsulate computation instructions (as in lines 12 – 22) or complex expressions. Another use case for functions is to access or modify structured data in a consistent way. Activities, on the other hand, typically maintain some state information either in their local variables or in their control flow and carry this state from one reaction step to another. This is useful when programming the mode switching logic of an application component.

Type extensions: Finally, let us turn our attention to the structure type defined in lines 6 – 33. In *Blech*, any data type may be extended by additional static code artefacts such as constants or methods. This example shows an extension with two methods: a function and an activity. The subtle difference between top-level activities or functions and extension methods is that methods always must be given a reference to an instance of the data type that they extend. For example, in line 12 the identifier[4] *this* is used to reference a given instance of the *PID* struct. A field of this instance can be accessed as in `this.KP`. If we were to allow a method to not only read but also modify the contents of an instance, then the method needs to be declared as a `mutating function` or `mutating activity`. This informs the causality analysis about the intended use of the passed instance reference. Code generation of methods is rather simple because they can be rewritten as normal functions or activities that simply receive an extra input or (when they are mutating) an extra output parameter. The *PID* structure shows how we can organise data and code in an object-based way. By design we do not support features of object-oriented programming such as inheritance, polymorphism through interface abstractions, or generics. The reason is that the first two, in their full generality, require the program to look up the correct implementation of a method at run time. This is known as dynamic dispatch and imposes too large a runtime penalty in an embedded real-time program. Generics would either require a runtime representation with the aforementioned drawbacks or we would have to generate monomorphic code for every instance of a generic data type (much in the fashion of C++ templates). The latter solution then raises questions regarding the intermediate representation of generic code to allow for separate compilation and also how to globally analyse and minimise the number of monomorphic instances. Instead we provide a set of built-in special container data types which are frequently needed, such as arrays, optionals, and events.

[4]It is not a predefined keyword as in Java but an arbitrary identifier as in F#.

Generally, note that in *Blech* all data is allocated statically. By design, there is no way to dynamically allocate memory and then deallocate or garbage collect it. In the same fashion, `cobegin` allows you to create an arbitrary but statically fixed number of concurrent branches. It is not possible to create "worker threads" based on some dynamic input. The reasons behind these decisions are that dynamic memory management is error prone and in the case of garbage collection unpredictable in terms of runtime. Moreover in a safety critical application one would expect to have a guaranteed memory bound after compilation. In practice, even with this memory bound, it is already challenging to ensure that the application meets its timing constraints. Finally, we believe that these features are not required by the type of embedded real-time application that we are targeting with *Blech*.

Causality: The PID example above uses concurrent composition of statements. However, in that particular example, the statements are unrelated in terms of data flow. One branch is observing whether a particular button has been pressed while the other branch deals with the calculation of controller commands from sensor readings. In general, it may be the case that the concurrent branches access shared data, which raises the question: in what order do they access that data and do they have a consistent view?

Assume we have two controller objects *pid1* and *pid2*. For the sake of argument, say that in each reaction the output of *pid1* is used as one of the inputs of *pid2*. This can be written in *Blech* as follows:

```
// assuming variables in1, in2, sp, out1, out2
     are in scope ...
cobegin
  run pid1:control(in1, sp)(out1)
with
  run pid2:control(in2, out1)(out2)
end
```

Our compiler will automatically deduce from the data flow of variable *out1* that in every reaction of this program, first a control step of *pid1* has to be performed, and only then a step of *pid2* is done. The lexicographic order of the `cobegin` branches is irrelevant for this sequentialisation, which means the following program behaves exactly the same.

```
cobegin
  run pid2:control(in2, out1)(out2)
with
  run pid1:control(in1, sp)(out1)
end
```

Using shared variables opens the door for two kinds of programming mistakes which both are automatically detected by the compiler. First, write-write conflicts:

```
cobegin
  run pid1:control(in1, sp)(out2)
with
  run pid2:control(in2, sp)(out2)
end
```

In the above example, both branches try to concurrently write to the same variable *out2* by mistake. This is forbidden. Compilation will stop and indicate the error to the programmer.

Another possible mistake is introducing circular read-write dependencies as in the next example.

```
cobegin
  run pid1:control(in1, out2)(out1)
with
  run pid2:control(in2, out1)(out2)
end
```

It is impossible to execute this program because *pid2* needs the output of *pid1* which in turn requires the output of *pid2*. Again, our compiler stops with a corresponding error message. Sometimes control algorithms do have such feedback loops but they are never instantaneous. There must be a known value, usually a value from the previous reaction, that can be used to start the current reaction. In *Blech*, this is expressed using the `prev` operator[5].

```
cobegin
  run pid1:control(in1, out2)(out1)
with
  run pid2:control(in2, prev out1)(out2)
end
```

This program tells the compiler to take the previous value for *out1*. Thereby the causality cycle is broken and sequential code can be successfully generated.

Automated causality analysis not only is a useful feature but also a guiding principle in *Blech*'s design. In the rest of the paper we focus on several of those design choices and explain them in more detail.

III. Key questions and related work

Defining when a given program shall be considered causally correct turns out to be non-trivial and gives rise to various notions of "constructive semantics" – a concise overview can be found in [11, Sect. 9]. Once a causality notion is fixed, related questions arise.

How can concurrent calls to subprograms be composed in a causally correct way?

Consider the typical example taken from [3] with two programs which are represented as sets of equations.

$$P : \forall n \in \mathbb{N} \left\{ \begin{array}{rcl} x_n & = & f(u_n) \\ y_n & = & g(v_n) \end{array} \right.$$
$$Q : \forall n \in \mathbb{N} \quad v_n = h(x_n)$$

The problem is whether P and Q can be composed concurrently. Alternatively, the program could be directly specified as

$$R : \forall n \in \mathbb{N} \left\{ \begin{array}{rcl} x_n & = & f(u_n) \\ v_n & = & h(x_n) \\ y_n & = & g(v_n) \end{array} \right.$$

The difference is that in R all assignments happen concurrently and obviously a causally correct scheduling can be found. However the concurrent composition of P and Q is not possible if P is sequentialised and compiled prior to composition with Q because then no causal order exists. Different approaches to separate compilation have been explored in literature:

Lublinerman et al. [7] propose a best-effort approach that "clusters" a subprogram into non-overlapping, concurrent parts. Thus the intermediate compiled code of a subprogram consists of precompiled parts and scheduling constraints among these parts. This allows the composition of precompiled subprograms by interleaving those precompiled parts while respecting the scheduling constraints and causality constraints. In this way, the problem above is solved, however the decomposition is not transparent to the programmer – changing the implementation may alter the compiler-generated decomposition and break existing software. Furthermore, the decomposition strategy requires the compiler to make trade-offs, e.g. between amount of code duplication and reusability, which are beyond the programmer's control.

Benveniste et al. [3] propose the extraction of an interface out of the code of a given subprogram. This interface is represented as an automaton with different kinds of relations between its states. Based on such interface descriptions it can be decided whether two subprograms are concurrently composable and how their individual actions need to be scheduled to maintain causal order. Thus the problem in the example above is solved in a similar manner as in [7] but with similar problems from a software engineering point of view.

In Quartz, compilation is the transformation of code to an intermediate format: the so-called synchronous guarded actions. Scheduling and causality analysis are not part of the compilation [5]. Calling subprograms in Quartz amounts to copying the corresponding code wherein all names (formal parameters) are substituted by the supplied arguments. Hence, there is no difference between R and the concurrent composition of P and Q from the example above. The drawback however is that causality is a global property and is only decided in a final code synthesis stage. Modular software development is impossible because there is no interface to program against.

How can structured data types be used concurrently?

Any program will usually make use of data structures commonly known as "structs" and "arrays". However in synchronous programming these structures become problematic when passed into concurrent subprograms.

Looking at the C code generator for Quartz we see that arrays are decomposed into individual variables that represent the cells of an array. The causality analysis is then straightforward but it also means that all array accesses must be evaluated at compile time. In other words: a for-loop running over an array is not implementable in Quartz.

In Scade [6], arrays permit only special, side-effect free operations such *map* and *fold* known from functional languages. Other operations are outside the language and have to be implemented in the host language. This raises the issue of dealing with foreign function code in the causality analysis.

Recently, Aguado et al. [1] proposed the use of a variant of interface automata to define admissible operations on an

[5]In Simulink, delays are used in the same fashion to resolve algebraic loops.

encapsulated data object. Their theory could be used to wrap, for example, arrays in an object that provides getters and setters and a policy that ensures a causally correct usage. From a language designer's point of view however we may ask whether this approach permits too much. If everyone may define an arbitrarily complex usage policy for any object, is it feasible to use third-party code and understand the potential error messages when used incorrectly?

IV. Causality

We propose acyclic schedulability as the causality notion that fits software needs. This results in simple programming rules:

- every variable is declared in the scope of a thread (which may read and write this variable arbitrarily once it is visible)
- upon a fork such a variable may be shared between the subthreads of which at most one may read and write it; the others may only read the variable and only after the last writing operation of the writer has finished in the current reaction step
- upon joining, the original (parent) thread reclaims all its access rights

This is a special case of the sequentially constructive semantics for synchronous languages [12] and can also be seen as a synchronous implementation of a causal memory model [2]. Our notion of a statically determined writer and potential readers aligns well with recent developments in actor based languages such as Rust[6] or Pony[7]. In today's embedded software the lack of clearly determined reaction steps, and writers and readers within such steps, is one of the main pain points in embedded software development. To our knowledge, Scade also only implements acyclic schedulers [6] but, being a functional or dataflow-oriented language, it does not permit sequential read-write operations as we do.

We now turn our attention to the individual questions being raised in Section III.

V. Separate compilation

A common theme to all papers mentioned in the introduction is that the causality interface of a subprogram is the result of some static analysis. Our approach goes in the opposite direction: we ban global variables entirely and use two parameter lists for our functions and activities: a list of input (read-only) parameters and a list of output (read-write) parameters. In this way, it is the programmer who defines a simple causality interface for his subprograms. Thus any program can be composed of pre-compiled subprograms which act as black-boxes and only declare a set of read-only and a set of read-write variables. On the basis of these interfaces alone, the compiler can check if a causally correct composition exists. This black-box approach enables us to treat causality interfaces as contracts in the same way as classical function interfaces: the programmer may alter the implementation as long as the new one abides by the same interface. The calling code never sees the change of the implementation. This is a crucial decision to allow for programming modular, maintainable and separately testable applications.

Reconsider the introductory example above: P and Q would be activities. Variables u and v are input variables, x and y are output variables for P. Thereby it is explicitly stated that in every reaction step both u and v are read to produce new values for x and y. Based on this information, our causality analysis will not allow the concurrent composition of P and Q. Indeed, if it really was the programmers intention to perform the two completely unrelated operations from P, he would be better off writing two activities: P_1 that computes x from u and P_2 that computes y from v. Then P_1, P_2 and Q could be composed – or the programmer implements R directly. Based on our experience so far, we believe that this simple, explicit, no-compiler-magic approach to subprogram interfaces is the most viable in the long run, especially for large projects.

VI. Structured data

We distinguish two kinds of data types: primitive (or atomic) and structured. Examples of atomic types are all numeric types or enumerations. They all represent a piece of memory that contains one value. Structured data types, on the other hand, represent a chunk of memory that is subdivided into smaller partitions. For example, structs are divided into fields that are accessible by name while arrays are subdivided into cells that are accessible by indices. Causality analysis is clearly defined for atomic types but structured types require us to make a decision: how fine grained should the causality analysis be? Consider the following (toy) example where control flow is forked into two concurrent branches which both try to write into shared memory:

```
struct Complex
  var real: float32
  var img: float32
end

function setReal(val: float32)(c: Complex)
  c.real = val
end

function setImg(val: float32)(c: Complex)
  c.img = val
end

  // ... in main activity ...
  var a: Complex
  cobegin
    setReal(-17.0)(a)
  with
    setImg(42.0)(a)
  end
```

From the interface of the functions *setReal* and *setImg* we know that the struct *c* may be modified. Hence calling these functions concurrently with the same output argument *a* results in a potential write-write conflict (and is hence forbidden).

[6]www.rust-lang.org
[7]www.ponylang.org

This is despite the fact that the two functions would write into disjoint memory locations. As with subprograms we take a black-box approach to causality analysis of data structures. So, by design, the causality interface does not specify which parts of a structured data type are read or written – it is always considered as a whole by the causality analysis. This ensures that the implementation does not "leak" into the interface. If we really want to concurrently write to disjoint parts of a structured data type then the caller has the responsibility of determining these parts. This means the callee must be designed to receive only the part it is writing to. We can rewrite the above example in this manner:

```
// Complex as before
function setValue(val: float32)(loc: float32)
  loc = val
end

  // ... in main activity ...
  var a: Complex
  cobegin
    setValue(-17.0)(a.real)
  with
    setValue(42.0)(a.img)
  end
```

This is a valid program. Admittedly, the function *setValue* is not very interesting in this example but in practice it could encapsulate some validation logic which we do not want to repeat twice.

While our semantics may be regarded as a trivial special case of the framework in [1], we believe this simplicity is a great advantage. Structured data can always be shared among any number of readers but is owned by only one writer as a whole. If different parts need to be written in reactions to different events, then either the writer needs to know about all these events and react to all of them accordingly, or the data structure needs to be disassembled into disjoint parts that are given to different writers where each reacts to one event only.

VII. References

Orthogonally to the notion of data being atomic or structured, we also discern between value- and reference-types. Note, that this has nothing to do with the way the data is handled in the generated C code. For instance, value-typed arrays will nonetheless be passed around functions using pointers; the distinction between reference- and value-types is a semantic one – not an implementation specific one. Primitive types, structs as well as fixed-sized arrays are value-types in *Blech* by default. Sometimes however, references to data are needed. Typical use cases include:

- Every formal parameter of a subprogram is a reference to a given argument.
- Aliasing of individual locations in a complex data structure, e.g.

  ```
  let ref rpm = wheels[3].rpm
  let ref rad = wheels[3].radius
  var ref speed = wheels[3].speed
  run rpmAsKmh(rpm, rad)(speed) // update
      the wheel's speed in every reaction
      given its rpm and radius
  ```

- Building data types that point to other data

  ```
  struct ValueStruct
    var a: int32
    var b: bool
  end

  ref struct RefStruct
    var x: float32
    var ref l: ValueStruct
  end
  ```

  RefStruct contains a reference to a location that contains a value-typed *ValueStruct*. As such, *RefStruct* itself is a reference-type, indicated by the `ref` keyword in its declaration. These *reference-types* are necessary to structure code in an object-based way.

References (i.e. aliases of other locations) as well as reference-type variables (i.e. instances of reference-typed data structures) must be assigned directly at their declaration and cannot be subsequently mutated. Note that their contents however may change throughout their lifetime – only the address is immutable.

```
var x: int8 = 5
var ref r = x
x = 17 // now also: r = 17
r = 42 // now also: x = 42
```

Immutability of references ensures that causality analysis remains decidable: we can statically determine the location that any given name points to. If we were to allow mutation, we would run into the undecidable problem of aliasing. The keywords `let ref` declare an immutable reference that may only read its contents while `var ref` declare an immutable reference that allows its contents to be changed. Our semantics require that `ref` is idempotent, i.e. a reference to a reference directly points to the original value, however the access capabilities may change. Continuing the example above:

```
let ref s = r // s points to x, but read-only
var ref t = s // compiler error: cannot grant
    write access to read-only location
```

A particular feature is that we have no dereference and accordingly no address operator. Depending on the context that a reference is used in, the compiler automatically generates the correct code that either accesses the contents of a reference type or passes the address it is pointing to. Consequently, whenever a reference is expected (say as a function parameter) in *Blech* we can pass a value typed variable or even a literal and the compiler will take care of finding the address or creating a temporary location.

Remember that in Section II we have introduced the `prev` operator to resolve causality cycles. It is important to note that this operator may only be used on value-typed data. The reason is that value-typed data (even structs or arrays) may efficiently be copied to store the previous value. References,

on the other hand, will usually define a tree of objects which we would have to walk over to create a deep copy. In order to exclude this performance pitfall, we generally have the rule: "no prevs on refs!"

VIII. Sharing data

So far, we have always assumed that an output parameter of a subprogram is distinct from any other formal parameter, i.e. the two names never point to the same memory location. This permits concurrent access to those parameters within the subprogram and also guarantees the programmer that writing an output parameter does not alter other parameters. On the calling side, this assumption restricts the caller since any two arguments must represent completely disjoint memory locations (unless they are both inputs). Consider the following activity which sums two numbers in every reaction and concurrently checks its second parameter for some threshold. When the threshold is exceeded, the weak branch is aborted and the whole cobegin block terminates.

```
activity add(a: int32, b: int32)(s: int32)
  cobegin
    await b > 10
  with weak
    repeat
      s = a + b
      await true
    end
  end
end

  // ... in main activity ...
  var sum: int32
  var x: int32; var y: int32

  run add(x, y)(sum) // OK, all distinct
  run add(x, x)(sum) // OK, overlapping inputs
  run add(x, sum)(sum) // error
```

The last call uses *sum* in both the input and output lists. This is not possible in general, because our separate compilation will compile the activity without knowing how it will be used. Thus the chosen sequentialisation of the `cobegin`-block may be fixed arbitrarily, in particular the check `b > 10` may be done before an iteration of the loop. This is not a problem as long as *a*, *b* and *s* are pointing to disjoint memory locations. If they do not, as in the last call, the threshold check is done on *sum* before a new value for *sum* is computed, which is not causally correct.

Sometimes however the programs are less restrictive. For example, consider an activity that just adds two numbers.

```
activity add(a: int32, b: int32)
            (s shares a, b: int32)
  repeat
    s = a + b
    await true
  end
end
```

Here it does not matter whether *a*, *b* and *s* represent the same location or not. The programmer explicitly declares this using the `shares` keyword. It is thereby guaranteed that these locations are not used concurrently inside the activity. Consequently, the call

```
run add(x, sum)(sum)
```

is allowed now. Sharing between parameters restricts how they can be accessed (concurrently) but it allows the passing of "overlapping" arguments. Extending causality analysis to respect the sharing annotations is straightforward.

IX. Ongoing and future work

The *Blech* to C compiler on which we are currently working is able to translate the basic control flow structures and atomic value types. Completing the remaining features is an ongoing process. Some of the work presented here forms the necessary prerequisite for an integration of *Blech* and C. It should be possible to call e.g. external C library functions directly from a *Blech* program. This will be a crucial step to enable the use of *Blech* in an industrial context.

X. Acknowledgement

A special thanks goes to Prof. Michael Mendler for his invitation to contribute to this special track of FDL 2018. We also thank the anonymous reviewers as well as Mark Andrew, Jens Brandt, Stephan Scheele, Matthias Terber and Simon Wegendt for their valuable feedback.

References

[1] Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha S. Roop, and Reinhard von Hanxleden. Deterministic concurrency: A clock-synchronised shared memory approach. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 86–113. Springer, 2018.

[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] Albert Benveniste, Benoît Caillaud, and Jean-Baptiste Raclet. Application of interface theories to the separate compilation of synchronous programs. In *Proceedings of the 51th IEEE Conference on Decision and Control, CDC 2012, December 10-13, 2012, Maui, HI, USA*, pages 7252–7258. IEEE, 2012.

[4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[5] Jens Brandt and Klaus Schneider. Separate compilation for synchronous programs. In Heiko Falk, editor, *12th International Workshop on Software and Compilers for Embedded Systems, SCOPES '09, Nice, France, April 23 - 24, 2009*, pages 1–10. ACM, 2009.

[6] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6: A formal language for embedded critical software development (invited paper). In Frédéric Mallet, Min Zhang, and Eric Madelaine, editors, *11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, September 13-15, 2017*, pages 1–11. IEEE, 2017.

[7] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 78–89. ACM, 2009.

[8] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, 2007.

[9] Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, Silvana Rossetto, and Adriano Branco. The design and implementation of the synchronous language CÉU. *ACM Trans. Embed. Comput. Syst.*, 16(4):98:1–98:26, July 2017.

[10] Klaus Schneider and Jens Brandt. *Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems*, pages 29–58. Springer Netherlands, Dordrecht, 2017.

[11] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2013. ISSN 2192-6247.

[12] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha S. Roop. Sequentially constructive concurrency - A conservative extension of the synchronous model of computation. *ACM Trans. Embedded Comput. Syst.*, 13(4s):144:1–144:26, 2014.