Chapter 2

Background: the ABS Language

The Abstract Behavioral Specification language[Johnsen et al., 2010a] (ABS for short) is a modeling language for concurrent systems. As such, it is well suited for describing, designing, and prototyping highly-concurrent computer software.

The ABS language is formally specified: the language's syntax and behaviour are not comprised merely of textual specifications or broader technical standards, but instead defined rigorously by means of mathematical methods. Since ABS is formally defined this makes it easier to analyze ABS models for possible deadlocks [Albert et al., 2014a, Albert et al., 2015b, Giachino et al., 2016b] or resource allocation [Albert et al., 2014a] and even fully verify properties over user-written functional specifications[Din et al., 2015]. Furthermore, the ABS formal semantics are laid out in a specific way that enforces the user to avoid certain problematic scenarios which arise during concurrent programming, such as race conditions and pointer aliasing.

ABS is executable — unlike other more "traditional" modelling languages — which means that any well-formed ABS model can be executed (evaluated) by a computer system. The ABS user can thus experiment and test any well-formed ABS model (e.g. by model-based test-case generation using symbolic execution [Albert et al., 2015c]) or even generate ABS code that can be integrated in production systems — currently there exist several ABS backends which generate production code partially or completely.

The syntax and programming feel resembles that of Java. In the rest of this section we introduce the basic elements and features of the ABS language in a manual-like style.

2.1 Data structures

All structures that hold data in ABS are immutable — with the exception of object structures, see section 2.3. An immutable structure cannot be updated in-place (mutated); instead the structure is copied into a new place in memory and its substructure updated. A common optimization is to not copy anew the whole updated structure but only its updated segment. Despite the obvious drawbacks of memory overhead and performance cost of copying, immutable data are considered beneficial in a concurrent but most specifically parallel programming setting for three reasons:

(a) Code can be written that does not have side-effects. This makes it easier for the user to *reason* about his/her program with the use of referential transparency (also known as equational reasoning) as well as the prover (human or not) to *analyse and verify* the code.

(b) Multiple threads can operate (i.e. read) the same location, but since the data does not change, the ordering in which different threads access it does not matter (no data races).

(c) The memory model becomes simpler; the compiler can thus apply code optimizations much more liberally.

The basic immutable data structures are the so-called *primitive data types* and consist of: Int standing for arbitrary-precision integers, Rat for arbitrary-precision rational numbers, String for (immutable) strings of Unicode characters. Integers can be implicitly converted to rationals (for more details, see section 2.4.2) but the other way around (downcasting) can only be done through explicit conversion (by using the function truncate), to avoid implicit (in other words, hidden) loss of precision errors in written ABS programs. All these primitive types are builtin inside ABS and cannot be redefined by the user or syntactically overwritten. Furthermore, there exist a special *builtin type* named Fut<A> which stands for a single containers of a value (of type A) that may be delivered sometime "in the future". For more about futures, see section 2.7. Since futures do not have a literal representation, they can be overridden. Example code of primitives and the special Fut is briefly given:

```
1               // Integer
1/1             // Rational
"text"          // String
obj!method(); // Futures created by async. method calls, see section 2.7
```

New user-written data structures can be given in the form of algebraic data types. Algebraic datatypes are high-level data structures defined as *products* and/or *sums* of types — types being other algebraic datatypes, primitives, and, in case of ABS, also of object types. A product groups together many data of different types, notated in set theory as $A * B * C \ldots * N$ where A,B,C,...,N are arbitrary types. Products resemble **struct**s in C-like languages and are denoted in ABS by:

```
data TypeName = ConstructorName(A,B,C,...,N);
```

where TypeName is the name of the type (required since ABS is statically-typed, see section 2.4) and ConstructorName is the name of the data constructor; in principle, a declaration of a data constructor name is not necessary unless the algebraic datatype contains also sums, but for the convenience of uniformity it is commonly required to give a constructor name for products as well. The most popular example of product types are *tuples*, with a triple of integers defined in ABS as:

```
data MyTriple = MyTriple(Int, Int , Int );
```

Sum types (also known as discriminated unions, tagged unions) groups together distinct types under a single "category" (type). The notation in set theory is $A + B + C + \ldots + N$ where $A, B, C, \ldots, N$ are arbitrary types (algebraic or object types) *as well as* product types. In other words, a sum type of $A, B, C, \ldots, N$ means that when a user "holds" a data structure with type $A + B + C + \ldots + N$, the contained value is of type either A, B, C, or N. The canonical example of a sum type is the boolean, given in set theory notation as $True + False$ and in ABS as:

```
data Bool = True
          | False;
```

where True,False are constructor names of their "nil-sized product types". The user could achieve the same in C-like languages with **enum** BOOL {false,true};. The extra power of algebraic datatypes shines when intermixing sums together with products; in set theory denoted by $(A*B)+(C)+(D*E*Z)+...$ (parentheses added only for clarity, strictly speaking they are unnecessary since $*$ takes precedence over $+$) whereas in ABS language:

```
data Type = Constructor1(A,B)
          | Constructor2(C)
          | Constructor3(D,E,Z)
          | ...;
```

Constructor names (e.g. Constructor(1,2,3) become important in sum types of statically-typed language since it allows us to *safely* (i.e. statically at compile-time) pattern-match on discrete values of possibly different, distinct types. Furthermore the contained types can be parametrically polymorphic with the use of type-variables:

```
data Either<TypeVar1,Typevar2> = Left(TypeVar1)
                               | Right(TypeVar2);
```

where TypeVar1 and TypeVar2 are type-variables standing for any possible type (algebraic or object type) which will be instantiated (be known) at use site.

The ABS language specification comes with a Standard Library that defines certain common algebraic datatypes such as Bool, Maybe<A> and List<A>, Set<A>, Map<A,B>:

```
export Bool, True, False;
export Maybe, Nothing, Just;
export List, Nil, Cons;
export Set,Map;
data Bool = True | False;
data Maybe<A> = Nothing | Just(A);
data List<A> = Nil | Cons(A,List<A>);
data Set<A> = //implementation;
data Map<A,B> = //implementation;
```

Note that Set<A> and Map<A> are so called *abstract* algebraic datatypes because their concrete implementation is not accessible outside of the module they are defined in (for our case inside ABS.StdLib). This is achieved by exporting only the types (i.e. Set, Map) and not the data constructors to the types, making them not accessible outside of the module. Abstract datatypes offer a two-fold advantage:

(1) operations on such datatypes preserve their invariants (e.g. no duplicate elements in a set or keys in a map, ordering, etc.) since the user cannot manipulate the data constructors of these types directly (by case-expression pattern-matching) but only through provided safe (in the sense of invariant-preserving) operations (functions).

(2) the individual (ABS) backends have the freedom to choose different purely (or not) functional datastructure implementations for those abstract datatypes.

2.2 Functional code

At its base, ABS adheres to a functional programming paradigm. This *functional layer* provides a declarative way to describe computation which abstracts from possible imperative implementations of data structures. Furthermore, ABS is said to be *purely* functional because inside any ABS program functional code cannot be mixed with side-effectful code (section 2.3). This pure/impure code distinction is achieved in ABS completely *syntactically*, compared to other purely functional languages where the same result is achieved at the type-system level (e.g. monads in Haskell).

At the centre of functional programming lies the *function* which is a similar abstraction to the *subroutines* of structural imperative programming, in the sense that it permits *code reuse*. However, unlike procedures, pure functions do not allow sequential composition (; commonly in C-style) since they completely lack side-effects. In the same manner, there is no need for an explicit **return** directive as the right-hand side evaluation of the function is the implicit return result (as it is mathematics). Note that sequential composition (;) is not the same as functional composition ($f \circ g$) because we are not composing right-hand side outputs of the functions but their underlying effects. The syntax of declaring an ABS function is:

```
def ResultType f<TyVar1,...TyVarN>(ArgType1 arg1, ..., ArgTypeN argN) = <expr>;
```

where f is the name of the function, arg1, ..., argN are the names of the formal parameters that the function takes (with their corresponding types) and ResultType is the overall type of the right-hand side expression. Furthermore, TyVar1, TyVar2, TyVarN are the typevariables that may appear inside formal parameters' types and/or ResultType. In this manner, functions can be parametrically-polymorphic, similar to to algebraic datatypes. Function definitions associate a name to a pure expression which is evaluated in the scope where the the expression's free variables are bound to the function's arguments. The functional layer supports pattern matching with a case-expression which matches a given expression against a list of branches.

An expression in ABS is either a primitive value (e.g. `1`, `1/1`, `"text"`), an applied data constructor (e.g. `Left(3)`), a fully applied function call (e.g.

f(True,4), ABS does not support partial application) an identifier (formal parameter or not), a case-expression, a *let-construct* or a combination of all of the above. A *let-construct* has the form let (Type ident) = <expr1> in <expr2> and binds the newly introduced identifier ident to point to expr1 inside the scope of expr2. The result-expression of a let-expression is the β-reduction of expr2 after capture-avoiding substitution of ident with expr1. The declared Type can be used to upcast the identifier if-and-only-if Type is a subtype of the expr1's actual type.

A case-expression is used to deconstruct a value of a datatype to its subcomponents and then assign particular identifiers to (some of) these subcomponents. This case analysis only makes sense for (non-abstract) algebraic datatypes, where the user has the ability to look inside the data constructors of the particular datatype. Other datatypes (primitives, abstract, algebraic, or object types) cannot be deconstructed and analyzed; only an identifier name can be assigned to them, similar to let-construct modulo the possible subtyping conversion. An example of the use of case-expression is given below:

```
def A fromMaybe<A>(A default, Maybe<A> input) = case input {
    Nothing => default;
    Just(x) => x;
    };
```

Each pattern => <expr> is a distinct branch of the case-expression. A (sub)-pattern can also be a wildcard (syntax: _) which matches any (sub-)component but does not bind it to an identifier. It should be mentioned that ABS does not do any case-pattern exhaustiveness search, which means that the ABS user can define partial functions, e.g.:

```
def A fromJust<A>(Maybe<A> input) = case input {
                      Just(x) => x;
                    };
```

which will throw a runtime exception (see section 3.2.1) when trying to evaluate fromJust(Nothing). Such data "accessors" are commonly used in functional languages, so the ABS language provides a shorthand for introducing such accessors (as syntactic sugar) at the point of the algebraic datatype declaration. For example, the above function will be implicitly defined, simply by annotating the constructor:

```
data Maybe<A> = Nothing
              | Just(A fromJust);
```

Finally, all primitive and algebraic data types provide default implementations for operations of (well-typed) structural equality (==) and lexicographical ordering (>,<,<=,>=).

2.3 Side-effectful and OO code

Keeping some form of state becomes handy when implementing certain algorithms, both for brevity and performance reasons. Stateful code also caters for C(++) and Java programmers, as the side-effectful and OO layers are much more familiar to them than the functional layer. ABS does not implement *stateful* computations through purely-functional abstractions (such as the State monad), but through the use of imperative programming (i.e. sequencing statements that possibly have side-effects). Furthermore, unlike an "observably-only" side-effect-free implementation of state (e.g. the ST monad of Haskell) ABS employs the full, *side-effectful* implementation of state as found in common imperative languages — in contrast, Haskell uses the IO monad. The reason that ABS uses side-effectful code is that albeit a modeling language, it allows certain observable communication with the real-world environment (e.g. println, readln, HTTP API) to facilitate user interaction during simulation (Chapter 4) or distributed computation (Chapter 5). As mentioned in section 2.2, ABS syntactically restricts the appearance of side-effectful code inside (purely) functional code. As such, side-effectful code can appear in ABS inside block scopes — a block is delimited by braces { } — i.e. the main-block (like the main procedure in C), every method-block (i.e. method body), while-block, if-then-else and if-then blocks.

The notion of local state in any imperative language is represented by *local variables*. Variables can be declared anywhere inside a method's body. The total scope of any variable is the scope from the start of its declaration line until the end of the current block. After declaration, they can appear inside expressions, be assigned and reassigned but not re-declared in the same or deeper scope. Furthermore, primitives (Int, Rat, String) and algebraic datatypes are forced to take an initial value, whereas object types and the special future type can be left uninitialized which will default them to null and unresolved future, respectively. An example of local variables inside a main block:

```
{ // main block can appear once per module

Int i = 3; // declaration/ initialization of a primitive
Maybe<Fut<String>> j = Nothing; // declaration/initialization of an ADT
```

```
i = i+1; // (re)assignment

Interf o; // declaration-only of an object type
Fut<String> f; // declaration-only of the special future type

j = Just(f); // (re)assignment

return Unit; // Unit returned by main, can be omitted

}
```

The main-block and every method-block can have a `return expr;` statement appearing strictly as the last statement of the block — this is too strict, since it would suffice to occur at every tail position so as to have a unique `return` point and no early exit of the method, but for clarity reasons the ABS language opted for a single-only return at the unique last position. If the `return expr;` statement is omitted, it defaults to `return Unit;` where unit is the singleton tuple (() in Haskell).

ABS is object-oriented: users can write classes which have a number of method definitions and fields. Fields can be declared in two positions:

```
class ClassName(<decl-pos1>) {
  <decls-pos2...>
  <method definitions...>
}
```

Fields at position-2 have the same initialization behaviour as local variables. Position-1 fields are instead left uninitialized and will be instead at creation time passed by the object creator as parameters (e.g. `new ClassName(params)`).

Fields can be referenced and reassigned inside any block with the prefix `this.fieldName`; fields have the same scope as their class. The special keyword `this` points to the currently executing object, much like Java. It is a syntax error for the main block to use the `this` or `this.fieldName` notation, since the main block lacks a `this`-object. An example of a class with one method block definition which adds to a field counter and returns the old counter value is given as:

```
class ClassName(Int counter) {
  {
      // init-block
  }
  Int addMethod(Int input) {
    Int oldCounter = this.counter;
```

```
    this.counter = this.counter + input;
    return oldCounter; // oldCounter is a local variable
  }
}
```

Instantiated fields and methods of an object are not visible by default outside its class scope. In practice this means that an object cannot access (read or modify) the fields of another object directly (all fields are private), but only through a method call, and any object can by-default only call its local methods (e.g. via calling this.m();). Calling a method of another object is achieved through explicitly exposed methods, which are bundled in interfaces. You can find more about interfaces and how they are used for (sub)typing in section 2.4.2.

Each class can have a single constructor, named the init-block. If omitted, it defaults to the empty-statement block. After the init-block finishes executing, the new object reference will be returned to the new-caller who can now resume execution with its next statement (i.e. new is a synchronous call). Also, after the init-block has finished, a Unit run() method will be implicitly asynchronously called for; this method is used for proactive concurrent objects. In Section 2.7 you can find more about synchronous/asynchronous calls and concurrency.

ABS lacks pointers and support for pointer-arithmetic. The evaluation strategy of ABS is strict, namely call-by-value semantics, much like Java where for primitive types, the value is passed, and for object types, the object-reference is passed as a value. ABS provides several common control-flow constructs: if−then−else branches and while-loops; there is no explicit **break**ing out of while loops. Any pure expression can be lifted to a side-effectful one. A case-statement, where case-branches associate to (side-effectful) statements, can be used instead of the similar but pure case-expression. Finally, ABS defines the equality operation (==) between objects to mean their referential equality; however, the ordering of (same-type) objects is left unspecified.

2.4 Type system

ABS is statically typed with a strong type system (strong referring to no implicit type conversion). The type system offers both System-F-like parametric polymorphism and nominal subtyping, commonly found in mainstream object-oriented languages.

2.4.1 Parametric Polymorphism

Parametric polymorphism appears in ABS in both datastructures and functions, e.g.:

```
data List<A> = Nil
        | Cons(A,List<A>);

def A head<A>(List<A> input) = case input {
  Cons(x,_) => x;
  };
```

The A above is a *type variable*, which means that it can take any concrete type at instantiation time.

Contrary to mainstream functional languages, the let-construct in ABS is non-recursive and parametrically monomorphic. Unfortunately, unlike other languages, there is no way to circumvent this monomorphism restriction, e.g. with an explicit type signature, since type variables in ABS can only be introduced either at data-structure or function definition and not in let definitions. For comparison, Haskell provides in addition to the explicit type signature approach, a language pragma to completely turn off the monomorphism restriction across the program modules.

Note that methods in ABS are parametrically monomorphic (compared to functions). Furthermore, there is no support for higher-rank parametric polymorphism since ABS lacks first-class functions to start with.

2.4.2 Subtype polymorphism

We saw in the previous section that the functions and algebraic datatypes (i.e. the functional core of ABS) are governed by a System-F-like type system: parametric polymorphism with no type inference. Instead, objects in ABS (imperative layer) are exclusively typed by interfaces. An interface, much like mainstream object-oriented languages, is a collection of method signatures. An example of an ABS interface is shown below:

```
interface InterfName1 {
  Int method1(List<Int> x);
}
```

A class is said that to implement an interface by writing:

```
class ClassName(params...) implements InterfName1, InterfName2... {
  Int method1(List<Int> x) { ... }
```

```
  ...
}
```

The ABS typechecker will make sure that the class implements every method belonging to the implements list of interfaces.

Unlike mainstream object-oriented languages, classes in ABS only serve as code implementations to interfaces and can not be used as types; as stated, in ABS an object variable is typed exclusively by an interface of its class, as in the example:

```
{
  InterfName1 object1 = new ClassName();
  object1.method1(Nil);

  InterfName2 object2 = new ClassName();
  ...
}
```

In the above example, object1 can be called only for the methods of its interface type InterfName1 and object2 only for InterfName2 accordingly.

Besides typing objects, the interface abstraction in many object-oriented languages serves also the purpose of nominal *subtype polymorphism* while ensuring strong encapsulation of implementation details. An interface type B is said to be a *subtype* of interface type A (denoted as $B <: A$) if it includes all the methods of A (and all of A's supertypes successively) and perhaps only adds new methods where their signatures do not interfere with any of the included methods (from the "supertype" interfaces). In ABS we have to explicitly declare that an interface is a subtype of another interface by using the extends directive, as shown in the following example:

```
interface  InterfName2 extends InterfName1 {
  Bool method2(Int y);
}
```

In other words we explicitly "nominate" InterfName2 to be a subtype of InterfName1 (hence the term nominal subtyping), by inheriting all of the InterfName1 methods (i.e. method1) and extending it with method2. This is in contrast to *structural subtyping* where we do not nominate the subtype relations of the interfaces but the relations are derived from what methods the objects do implement (i.e. their structure). For example, under structural subtyping if an object o1 implements two methods m1 with type t1, m2 with type t2 and object o2 implements only m2 with type t2, then object o1's overall type is a subtype

of o2's overall type, thus o1 can be safely upcasted to o2's type. The main benefit of structural subtyping is that it makes it possible to infer the overall types of the objects, but it comes with the drawback of accidental subtyping (upcasting), when there exist methods among objects with same signature but different "purpose". With nominal subtyping, accidental upcasting does not occur since the user provides explicitly the subtyping relation during interface declarations. An example follows of the (implicit) upcasting in ABS:

```
InterfName2 o = new ClassName();
o.method2(3);

InterfName1 o_ = o; // upcasting to super interface if InterfName2<:InterfName1
o_.method1(Nil);// can only call method1, method2 is not exposed through object o_
```

Note that, besides object types (typed by interface), the primitive types Int and Rat, albeit not represented through (mutable) objects, are associated by a subtype relation as well, where Int is a subtype of Rat, i.e. $Int <: Rat$.

2.4.3 Variance

Combining parametric polymorphism with (nominal) subtyping leads to the overall type system of ABS. Two important questions that arise in such a type system is a) what is the default *variance* of the abstractions offered by the language and b) is the user able to manually (as in syntactically) change their variance.

Generally, there are three different notions of variance:
Assuming B subtype-of A, i.e. $B <: A$,

(i) An abstraction C is *covariant* iff C<B> <: C<A>.

(ii) An abstraction C is *contravariant* iff C<A> <: C<B>.

(iii) An abstraction C is invariant if it cannot be further subtyped: neither (i) nor (ii) hold.

For certain abstractions, there are sensible variance defaults. E.g. immutable algebraic datatypes can be covariant by default, and pure functions are contravariant in their input types and covariant in their output type. There are reasons, however, that a user wants to change or restrict the default variance of an abstraction, e.g. a user wants to make an abstraction invariant because they know that the abstraction does not have to be later subtyped, or

the implementation of the abstraction poses certain restrictions which deem it invariant.

The standard ABS type system [Johnsen et al., 2010a] (given as type rules of type theory) does not completely specify the default type variance (a). Furthermore, when ABS uses the term "subtyping" it refers to the common-notion of width subtyping and not that of depth subtyping[1] Taken from the specification of the ABS language:

> $T <: T$ is nominal and reflects the extension relation on interfaces. For simplicity we extend the subtype relation such that $C <: I$ if class C implements interface I; object identifiers are typed by their class and object references by their interface. We don't consider subtyping for data types or type variables.

So it is left to the particular ABS compilers to define their support for the variance of ABS abstractions. Many compilers (Maude-ABS, Erlang-ABS, HABS) provide sensible defaults of covariant subtyping for algebraic datatypes with only for width subtyping which is the default subtyping we described in this section (not depth subtyping). Finally, there is no current syntactic extension to the ABS language to provide means for manually changing the variance of user-written code.

2.4.4 Type Synonyms

Standard ABS provides language support for type synonyms. A type synonym of ABS is an "alias" assigning a (usually shorter, mnemonic) distinctive name to an algebraic datatype, object type, type synonym, or a combination of those. An example of a type synonym in ABS is shown below:

```
type CustomerDB = Map<CustomerId, List<Order>>;
type CustomerId = Int;
type Order = Pair<ProductName, Price>;
type ProductName = String;
type Price = Int;
```

[1] The term depth subtyping quite differs in meaning than the commonly found (width) subtyping. For a general description of what is depth subtyping, see `https://en.wikipedia.org/w/index.php?title=Subtyping&section=5#Width_and_depth_subtyping`

2.5 Module system

The ABS language includes an elaborate *module system*, inspired by that of Haskell. Modules can be specified in the same file or in separate files. Each module has at most one main block. The ABS user decides which main block will be the entrypoint of the program at the compilation step. Furthermore, by not exposing some or all data constructors of an algebraic datatype, the ABS user can designate the datatype to be *abstract*, i.e. it hides its concrete internal implementation. An example of the different constructs of the ABS module system follows:

```
module MyModule; // the beginning of a new module

export D,f,x; // exports specific identifiers
export * from M; // exports everything of imported module M
export *; // exports all local and imported identifiers

import M.ident; // imports identifier from module M as qualified
import ident from M; // imports identifier from module M unqualified
import * from M; // imports all exported identifiers of M unqualified
```

2.6 Metaprogramming with Deltas

Class inheritance, also known as code inheritance, is abolished in favour of code reuse via delta models [Clarke et al., 2010]. A delta can be thought of as a non-line-based patch (generated by Unix diff program) or better even, as a higher-level C macro. Unlike common preprocessors that check only for syntactic errors of the macros applied, deltas can also be checked for semantic errors, i.e. if certain delta applications are invalid. An example of delta metaprogramming in ABS, taken from [Gouw et al., 2016], follows:

```
delta RequestSizeDelta(Int size); // name of the delta
uses FredhopperCloudServices; // which module to apply on
modifies class ServiceImpl { // modifies class
  adds List<Int> sizes = Nil; // adds field
  modifies Bool invoke(Int size) { // modified method
    sizes = Cons(size, sizes);
    return original(size); // uses original code
  }
}
```

Software product lines can be conveniently realized through feature and delta models (i.e. groups of features, and groups of deltas; deltas implement the features) [Clarke et al., 2010]. Specific software products can then be generated from the product line by selecting the desired features, as shown briefly below:

```
productline ProduceLine;
features Request, Customer;
delta CustomerDelta(Customer.customer) when Customer;
delta RequestSizeDelta(Request.size) when Request;

product RequestLatency (Request{size=5});
product OneCustomer (Customer{customer=1});

root Scaling {
  group [1..*] {
    Customer { Int customer in [1 .. 3]; },
    Request { Int size in [1 .. 100]; },
  }
}
```

2.7 Concurrency model

The foundation of ABS execution derives from the actor model [Hewitt et al., 1973]. The actor model is a model of concurrent computation where the primary unit of concurrency is the actor. An actor system is composed of (many) actors running concurrently and communicating to each other unidirectionally through messages. Unlike other well-known models for concurrent computation, the actor model arose "from a behavioural (procedural) basis as opposed to an axiomatic approach" for example that of Milner's π-calculus and Hoare's CSP.

Although the actor model is well-studied and discussed, there is no wide consensus on what the actor model consists of and what not. Furthermore, for practicality or implementation reasons, widely-used actor software deviates from the original Actor model specification. Arguably, the closest software implementation to the Actor model currently can be found in the Erlang programming language. For this reason, most of the following actor code examples are represented in Erlang's syntax. What follows is a rough list of the key properties found in the Actor model:

- Share-nothing philosophy where actors have private state and do not share memory with each other, but communicate only and explicitly by messages.

- Sending a message to another actor is *asynchronous*. The message will be put in the receiving actor's *mailbox*. To receive a message, the actor picks a message from its mailbox, an operation which is (usually) *synchronous*.

- After receiving a message, an actor has the choice either to stop executing, modify its private state, create new actors, send messages or decide (at runtime) to receive a different message (i.e. change dynamically its behaviour).

- There is no pre-defined ordering of message delivery: specifically, no local ordering that dictates that the messages of an actor arrive in the same order they were sent by that actor; nor is there a global ordering where sending actors can prioritize their own message over other actors.

- Actors are uniquely — across the whole actor system — addressable. An actor's address becomes known to other actors either upon actor creation or when an actor explicitly exposes its own address (commonly named **self**, which can be thought as OO's this):

```
% creates a new actor to run function(args). Returns the new actor's address
OtherActorId = spawn(function, [args]),
% sends its own actor address (self) to another actor as a message
OtherActorId ! (self, payload).
```

The concurrent execution model of ABS is the result of combining the object-oriented paradigm with the actor model. Specifically, on top of the synchronous method calls of (passive) objects of OO languages, ABS adds support for inherent concurrency and asynchronous communication between such objects: the result is called an *active object*.

The active object (also often named concurrent object) is based on the usual object found in mainstream OO languages, with object caller, object callee (this in ABS) and synchronous method call (callee.methodName(args)). Influenced by the actor model, the active object is extended with a mailbox for receiving messages. As in the actor model, there is no defined message arrival ordering inside the mailbox. Unlike the actor model, messages in ABS are not arbitrary (and possibly untyped) atoms, but instead type-checked methods that the callee explicitly exposes (via interfaces). Sending such a method

(as a message) is accordingly named making an asynchronous method call (callee !methodName(args)).

```
object . method(args); // synchronous method call
object ! method(args); // asynchronous method call
```

A further deviation from the actor model is that the communication between ABS active objects is by default two-way whereas using the actor model we would need two (unidirectional) messages: a message with the request payload (plus the **self** identity) and a response message to "self" actor, plus the response payload. In active objects this is encapsulated inside the method's definition: the request payload are the method's actual parameters and the response payload is the return value.

```
main() ->
  Actor = spawn(className,[]),
  Actor ! {method,args,self}, % make an asynchronous method call
  ...
  receive
    Response -> doSomethingWithResponse(Response)
  end.

className() ->
 receive
   {method,Args,Sender} => Response = method(Args),
                           Sender ! response()
 end.

method(Args) = <impl>
```

```
{
  actor = new ClassName();
  actor ! method(args); // no need to send self (or this)
}

class ClassName() {
  ResponseType method(<args>) {
      ResponseType response = <impl>;
      return response; // the response is sent when return is called
  }
}
```

Another difference is that this two-way communication is a first-class citizen of the ABS language, called *future* and represented as Fut<A>. Upon

establishing the communication, a future is created and assigned a unique identity among the active-object system. In the simple actor model (e.g. Erlang) a future abstraction has to be manually implemented by perhaps some unique tagging.

```
{
  actor = new Class();
  Fut<ResponseType> future1 = actor ! method(args); // asynchronous method call 1
  Fut<ResponseType> future2 = actor ! method(args); // asynchronous method call 2

  Bool b = future1 == future2; // FALSE identity comparison
  ...
  ResponseType response1 = future1.get; // block until response is ready
  doSomethingWithResponse(response1);
}
```

Get-blocking operation Holding a future is similar to holding a non-blocking reference to the "future" result value of an asynchronous method call. Instead, reading this future value (futureReference.get) is an operation which will block until the asynchronous method call has finished and the result has been communicated back. Futures are not restricted only to the caller but can be passed around and read from other objects; however, futures are written only once and only by the callee object. Futures can be tested in ABS for equality (==) based on their assigned identity (referential equality). The standard of ABS does not define a specific ordering on futures.

An ABS system is comprised of active objects executing their actions *concurrently* between each other, i.e. sending messages (method calls), receiving messages (method calls), sending responses (return), waiting for responses (get). As in the actor model, the level of concurrency (scheduling/interleaving) between active objects (actors) is left unspecified — although it usually assumes some starvation-freedom guarantees.

The ABS language adds an extra abstraction on top of active objects: the option of *grouping* active objects together. Every active object strictly belongs to one such group (named Concurrent Object Group, COG for short). To create an active object and put in a brand-new COG, the user uses the expression new, whereas to create an active object inside the current COG the expression new local:

```
InterfName object1 = new ClassName(params); // new object in a new COG
InterfName object2 = new local ClassName(params); // new object in current COG
```

In the simplest case where each active object lives in its own COG (through using only `new`), the same as before holds where the active objects in the system are executed concurrently (preemptively scheduled to avoid starvation). When forming larger groups (size > 1), each COG will essentially be a preemptively scheduled entity, whereas the active objects inside each COG will be *cooperatively scheduled* for concurrency. By "cooperation" we refer to the *intra-object* (i.e. inside the same COG) scheduling of ABS processes and not the synchronization between objects (inter-object communication), which is achieved through the previously describe `get` operation.

Cooperative scheduling means that an active object can decide to deliberately (syntactically) yield control to another object of the same COG. In other words, a COG can be seen as an individual (as in independent) processing unit where *at most one* active object is running on. Active objects on the same COG do not share their mailboxes, but share their processor (COG) for resources in cooperation. An active object "cooperates" by deciding to "pause" its execution and give the processing resources to any other object of the same COG. An active object may be later given back the resources to resume execution — same as a *semi-coroutine*, also called *generator*.

This cooperation (yield of control-execution-resources) manifests in ABS in three distinct forms:

Yielding unconditionally. The `suspend` statement releases control of the currently executing active object to some other object of the same COG (including possibly itself).

Awaiting on futures. A statement with the form `await futureRef1? & ... & futureRefN?;` means that the current object yields control and will not be resumed at least until all the given futures have their values ready. In contrast to `futureRef.get;` this does not block the whole execution unit (COG).

Awaiting on booleans. A statement of the form `await booleanExpr1 & ... & booleanExprN` means that the current object yields control and will not be resumed at least until all the boolean expressions evaluate together to `True`. Boolean awaiting makes only sense when used with boolean expressions that can change, i.e. contain some object fields, e.g. `await this.x==this.y*2`.

The arguments of the latter two forms can be combined with the operator (`&`), which basically means that the active object decides to yield at least until

the specified futures have been resolved and the boolean conditions evaluate to True at the same time, e.g. await fut1? & this.x>3 & fut2? & this.x==this.y*2.

Since it is common practice to write code that includes an await on future following by its get, the ABS provides some syntactic sugar:

```
{
 A result = await o!m(args);
 // is syntactic sugar for
 Fut<A> hygienicRef = o!m(args);
 await hygienicRef?;
 A result = f.get;
}
```

It is worth mentioning that although the ABS standard leaves the preemptive as well as the cooperative scheduling underspecified, many ABS backends employ some concrete strategy, whereas ABS analysis and verification tools may explore many (if all) schedulability options.

Even with scheduling of messages being open to interpretation, the structure of the ABS language and its concurrency model avoid common problems that arise in concurrent programming, such as race conditions, deadlocks and pointer aliasing. First, the immutability of datastructures serves as the ground base to avoid a lot of race conditions that commonly arise in imperative programming. Moreover, futures which are commonly shared between objects (and their processors/threads) are also write-once (immutable). Secondly, the fields of any objects are not directly exposable to other objects, which leads to less incidents of pointer aliasing. Finally, the scheduling points in ABS are always explicit (i.e. suspend or await) which makes it easier to reason about the possible interleavings of an ABS program.

Note. The old ABS standard specified that synchronous method calls caller.m(args) where the caller and the callee belong to different COGs is a runtime error. The new version of ABS allows such synchronous method calls between different COGs, and translates them to the sequence Fut<A> hygienicRef = caller!m(args); hygienicRef.get;. This is not semantically equivalent to the synchronous method call of OO languages (also for same-COG objects), because although the caller blocks during this call, the callee does not "immediately" execute the corresponding method body; instead, the method is put in the mailbox to be later (undetermined and thus not immediate) executed.

2.8 History of ABS

The ABS language has its origins in the Creol language which is in turn the continuation of the SIMULA language. The Creol language [Johnsen et al., 2006] had features that the current (as of 2017) ABS standard has since abolished, e.g. cointerfaces (interface typing for callers as well) and class (code) inheritance (replaced instead with Delta metaprogramming). The current ABS enhanced the initial Creol language with support for algebraic datatypes with parametric polymorphism and pattern matching, pure expressions, and Concurrent Object Groups, inspired by the JCoBox Java extension [Schäfer and Poetzsch-Heffter, 2010].

The ABS language has been syntactically and semantically evolved through three successfully-completed European projects:

1. CREDO: "Modelling and analysis of evolutionary structures for distributed services". 2006–2009, `https://projects.cwi.nl/credo/indexpub.html`
2. HATS: "Highly Adaptable and Trustworthy Software using Formal Models". 2009–2013, `http://hats-project.eu`
3. ENVISAGE: "Engineering Virtualized Services". 2013–2016, `http://envisage-project.eu`

2.9 Comparison to other concurrent, modeling languages

The most well-known modeling language is the *Unified Modeling Language* (UML). Albeit a general-purpose modeling language, UML focuses on the design of software systems (mostly object-oriented-based), similar to ABS. Unlike ABS, UML is not in general executable, although there have been certain tools (Microsoft Visual Studio, Eclipse IDE) that can generate program code which derives from UML models. Moreover, UML is defined by standard committees (Object Management Group, International Organization for Standardization) and is not defined (as in the case of ABS) using rigorous formal-methods procedures. UML support for modeling concurrency (through means of interaction diagrams, e.g. sequence diagram, communication diagram, interaction overview diagram) is arguably not adequate to capture the interdependencies, evolution over time (creating/destroying actors) and inherent indeterminacy (unbounded non-determinism) of concurrent as well as distributed systems.

The Process Meta Language (PROMELA) of the SPIN model checker [Holzmann, 2003] is an executable modeling language used to verify (by means of model-checking) concurrent software systems. The concurrency unit in PROMELA is the process; the execution of processes is interleaved (unless using atomic blocks) and the communication between processes is achieved through (buffered) channels which can be globally shared. Also PROMELA has a "choice" construct to express bounded non-determinism. In those respects, PROMELA's model of computation resembles more that of Communicating Sequential Processes (CSP), than the actor model. Furthermore, the language has, justifiably, limited support for complex data types, since "larger" datatypes is one of the culprits for the problem of state explosion during model checking.

Rebeca ([Sirjani et al., 2004]) is a verifiable modeling language which combines as well the object-oriented paradigm with the actor model. If the number of "active objects" and size of their mailboxes are bounded, Rebeca can generate models in lower-level code so as to apply model checking by using external model checkers, e.g. SPIN. Rebeca also encapsulates the message passing of the Actor model behind the (asynchronous) method calls, but the communication is one-way, i.e. such asynchronous methods cannot return values. Since there are no such "responses" to asynchronous method calls, there is also no need for futures and their awaiting mechanism. In other words, the active objects of Rebeca run in an interleaving manner between them, but their methods only "appear" to run atomically, meaning that each method is executed from start to end; there exist no mechanism (suspend, await) to jump midway the method's body to a different method execution of the same object. Finally, there is no support for algebraic datatypes and COGs, but there is syntax for representing bounded non-determinism (choice).

Eiffel (`http://eiffel.org`) is one of the first statically-typed object-oriented languages, and became known for its design by contract philosophy, i.e. associating invariants to classes and contracts in the form of pre- and post-conditions around method bodies. Unlike ABS, these conditions will not be checked statically, by analysis and verification tools, but only at runtime (like the familiar assertions). The language provides limited support for concurrency, mainly by means of traditional threading, but like ABS offers private-only fields and limited form of parametric polymorphism. In a perhaps similar spirit, the Java Modeling Language (JML) tries to introduce a lightweight method of design-by-contract and formal verification to (existing) Java object-oriented codebases.

A more detailed discussion of real-world concurrent programming languages and runtimes can be found in section 3.9.