Semi-Dynamic Session-Types for ABS

Bachelor thesis in Computer Science by Anton Wolf Haubner
Date of submission: 2019-11-08

1. Review: Prof. Dr. rer. nat. Reiner Hähnle
2. Review: Eduard Kamburjan, M.Sc.
Darmstadt – D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Software Engineering Group

Erklärung zur Abschlussarbeit
gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Anton Wolf Haubner, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den 2019-11-08

A. W. Haubner

Contents

1. Introduction

Distributed systems are omnipresent in the modern computing world. Their application ranges from the small-scale, e. g. phones interacting with smart home devices, to the processing of big data, which encompasses gathering information from distributed sites and analyzing it in clusters.

Reasoning about distributed systems is difficult due to their concurrent and asynchronous nature [10] and can incorporate a variety of questions: “Is the system deadlock free?”, “What side-effects does an interaction have?” or “Does the system follow a certain communication protocol?”. This thesis focuses on protocols which describe the structure of interactions in a distributed system of active objects. This for example includes the order of asynchronous calls between objects of the system, at what point a result has been computed, a process been suspended or continued etc.

Manually checking a model of a distributed system against a protocol specification quickly becomes too labor-intensive and time-consuming. Thus, the need for automatic verification arises, which can reduce the workload and help to avoid human errors.

However, in a system where messages can arrive out of order, static verification is not always enough to guarantee a system’s behavior follows a protocol. Also, if not all parts of a system are previously known or behavior can be added during runtime, then these unknown components may interfere with a protocol.

Our solution to these problems is to dynamically enforce protocol adherence at runtime. This is achieved by adding schedulers to the objects of a system which allow only those tasks to execute that do not violate the protocol.

1.1. Contribution of this Thesis

A concept for statically verifying the structure of communication between objects of a distributed system and also enforcing it at runtime has been developed by Kamburjan, Din and Chen in [30] for the ABS language [27].

This thesis aims to provide a first implementation of the concept by developing a tool on top of the ABS compiler [44]. This tool takes an ABS model of a distributed system and a protocol specification as input. The protocol specification is checked for errors using a new method which has been inspired by *Configurable Program Verification* (Beyer, Henzinger and Théoduloz [4]). The tool then statically verifies that each participating object of the model locally conforms to the protocol specification. The model is also extended to enforce the specification at runtime. The checked and extended model is then compiled by the ABS compiler, which required minor adjustments to its Erlang [34] backend.

Furthermore, the thesis incorporates ideas from [29], which extends the protocol specification to allow reasoning about side-effects of interactions. The tool developed during this thesis verifies the specified side-effects at runtime.

Additionally, case studies are conducted to evaluate the concept. They answer questions about the performance impact of the dynamic enforcement. Also, they are used to gain confidence that our tool can reliably detect violations of a protocol and correct behavior at runtime.

We call our developed tool in the remainder of this thesis the "*Semi-Dynamic Session Type Tool*", or SDS-tool in short.

1.2. Structure of the Thesis

Chapter 2 introduces some theoretical concepts and background information this thesis is based on. This includes a definition of distributed systems, a description of *session types* on which the specification language of protocols is based on and a short introduction to the ABS language, which is the working environment of the developed tool.

In Chapter 3, the underlying concepts of validation, static verification and dynamic enforcement of a protocol specification are elaborated on and discussed whereas Chapter 4 presents noteworthy implementation details.

Chapter 5 describes the case studies conducted to evaluate the resulting tool and reviews the results.

Finally, Chapter 6 summarizes the thesis's contents and results. It also contains a short survey on related work and possible extensions of the thesis product.

1.3. Notation

Powerset We write $2^S = \{S' \mid S' \subseteq S\}$ for the powerset of set S.

Functions The set of all total functions mapping elements of set S to set S' is denoted by $S \rightarrow S'$. The set of all partial functions from S to S' is denoted by $S \rightharpoonup S'$. $f : S \rightarrow S'$ is an alternative notation for $f \in S \rightarrow S'$.

For the *updated* function of f where the output value of f is replaced with b for an input a, we write $f[a := b]$:

$$f[a := b](x) = \begin{cases} f(x) & \text{if } x \neq a \\ b & \text{otherwise} \end{cases}$$

If we want to update multiple output values in $f : A \rightarrow B$ we write $f[S]$, where $S : A \rightharpoonup B$ and

$$f[S] = (f \setminus \{(a, b') \mid (a, b) \in S \land b' \in B\}) \cup S$$

.

Abbreviations We sometimes replace parts of formulae and terms with the placeholder symbols "…" and "·" to abbreviate them if the concrete value of the replaced part is of no special importance to a statement. In logical formulae, each occurence of "·" may be rewritten with a previously unused, universally quantified variable. For example,

$$\exists x. \, (x, \cdot) \in R$$

can be rewritten as

$$\forall y. \, \exists x. \, (x, y) \in R$$

.

Grammars We notate grammars in the Extended Backus–Naur Form. Nonterminals are enclosed in a box. Optional parts of a production are enclosed in square brackets “$[T]$”. Parts which can either be omitted or repeated are annotated with a bar “$\overline{T}$”.

$$\boxed{\text{nonterm}} \quad ::= \quad \boxed{\text{nonterm}} \;\mid\; \texttt{term}\,[\texttt{optional}]\,\overline{\texttt{repeatable}}$$

In general, we will use “$[\ldots]$” and “$\overline{\ldots}$” to mark optional or repeatable parts of formulae, terms or code listings.

2. Background

2.1. Distributed Systems

Tanenbaum and Van Steen loosely define distributed systems in the following way [47]:

> A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

From this definition two characteristics of distributed systems arise:

1. There must be computing elements which are independent of each other. These can, for example, be a number of processors connected by a network.
2. Ideally, these computing elements must interact with each other in a collaborative way so that the user of the system does not notice that the system is distributed. For our purposes, however, it is enough to require from a distributed system that there are means of communication between the computing elements that allow them to collaborate.

Now, there are different ways of abstractly modeling a distributed system, for example, with varying computing elements (processes, actors, ...) or modes of communication (message passing, shared memory, ...). Since the tool developed in this thesis operates on models described in the ABS language, we adopt a characterization of distributed systems based on active objects which is compatible with the concurrency model of ABS.

2.1.1. Active Objects

Active objects [7] are one possible model of computing elements in a distributed system and a special case of the actor model [1] of concurrent computation.

Actors encapsulate an internal state and communicate via message passing. Computations within an actor are always triggered as a response to a message. A computation can then send messages to other actors, create additional actors and change the internal state of its own actor to influence future behavior.

In the case of active objects, each active object defines the syntactic instructions for its computations as a set of methods. Messages are implemented as asynchronous method calls between the objects. Thus, active objects adjust the actor model to conform to the object-oriented programming paradigm, making it more accessible to programmers familiar with this paradigm and suitable for practical use in object-oriented languages like ABS.

Please note, that we are using the terms *active object* and *actor* interchangeably in the remainder of this thesis, since we are exclusively reasoning about distributed systems where all actors are active objects. We denote the set of all active objects by *Actors*. Variables p and q usually range over *Actors*.

2.1.2. Model of Concurrency

Active objects execute concurrently, but within each active object, there is only one thread of control. Also, since communication is asynchronous, the receiving actor of a method call does not need to explicitly expect communication, for example by busy-waiting for input on a channel.

This means, that if an active object is currently executing a computation, received asynchronous method calls are buffered until the current computation concludes. Only then can the next method be executed, occupying the single thread of control of the active object.

Likewise, when issuing an asynchronous call, execution in an active object simply continues without waiting for the result of the call. Instead, the asynchronous call produces a future value (see below), which can be used to explicitly synchronize with the conclusion of a call and to retrieve its result as soon as it is available.

Futures

Futures are a means of referencing the result of an asynchronous computation which may not have finished execution yet.

There exist slightly different definitions of futures [6, 15], however, in the context of active objects, a future is a placeholder value received upon calling a method asynchronously on an object. In this context, a future is used primarily for two purposes:

1. to synchronize with the completion of the asynchronous computation referenced by the future.
2. to retrieve the result of the asynchronous computation after it completed.

An explanation of the usage of futures as a feature of the ABS language is available in Section 2.2.4. In the remainder of this thesis, we denote the set of all futures as $\overline{\mathcal{F}}$. The set of symbols representing futures is denoted by $\mathcal{F}$. Variables f, f' etc. usually range over $\mathcal{F}$ if not specified otherwise.

Cooperative Scheduling

Active objects can also support cooperative scheduling, as does ABS. This means that they allow the interleaving of method executions, but only at explicitly marked suspension points of a method. For example, those points are usually employed to wait until a future can provide the result value of a call.

Therefore, due to cooperative scheduling, active objects select the next computation to execute from a pool of not only asynchronous method calls, but also from suspended method executions that can be continued.

This process of selecting a next computation and executing it is from now on called a *method activation*. We call the initial activation of a method for a call a *method invocation*. If an execution is activated again after it has been suspended, we say it is *reactivated*.

2.1.3. Non-Determinism

The order in which messages (method calls) arrive at an active object is arbitrary, as is the order the active object will execute them on its single thread of control. This introduces a source of non-determinism to a distributed system.

ABS gives the programmer the opportunity to gain back some control by allowing them to optionally define so-called scheduling functions (see also Section 2.2.4). These functions can deterministically choose the next method to execute from the currently possible (re-)activations. The developed SDS-Tool makes heavy use of this feature of ABS to enforce method ordering at runtime.

2.1.4. Dynamic Topology

As mentioned before, active objects may create other active objects. They may also communicate references to active objects, which is like an address, messages (method calls) can be sent to.

Therefore, a distributed system of active objects can at any time extend itself with new active objects and reconfigure its topology.

2.1.5. Fairness

Agha [1] argues, that the weakest form of fairness in an actor based distributed system is the guarantee of delivery of communications, that is, that every asynchronous method call in a system of active objects will eventually be executed. However, since ABS allows for arbitrary scheduling functions there is no such guarantee in our concept of a distributed system.

Moreover, originally ABS does at least guarantee, that a method will be activated if the active object is idle and method activations are available. But because our dynamic enforcement of a protocol sometimes requires execution to wait until a specific method activation is available, this behavior raises a problem. Therefore, ABS had to be modified during this thesis, such that scheduler functions are able to refuse to select a method activation, see Section 3.5.5.

Thus, there is in general no guarantee that any buffered activation in an active object will be executed.

2.2. ABS language

ABS, the “**a**bstract **b**ehavioral **s**pecification” language, has been designed for the purpose of modeling object-oriented, distributed systems [27].

ABS tries to keep a balance between allowing to formalize high-level specifications of such systems and the detailed description of their behavior so that the resulting models are still executable.

Since the syntax, a type system and operational semantics have been formally defined for ABS, it is also well-suited for developing formal methods tools, like verification software [20]. Because this thesis is concerned with the development of a tool for verifying interactions and enforcing protocols in a distributed system, ABS is thus a fitting implementation environment.

2.2.1. Layered Architecture

ABS is divided into a set of layers called *Core ABS* and a set of independent extensions. Their combination is labeled *Full ABS*.

The layers of *Core ABS* each implement a different language concept, see Figure 2.1. For example, the lower 4 layers contribute a programming language which encompasses object-oriented, imperative and functional concepts and that is quite similar to Java. Among other things, the remaining layers add on a concurrency model.

The following sections inform about those layers that are relevant to the thesis in more detail. This mainly includes the concurrency model and a brief description of the programming language.

2.2.2. Functional Layer

The functional layer of ABS consists of algebraic data types and first-order, as well as second-order functions, called partially defined functions.

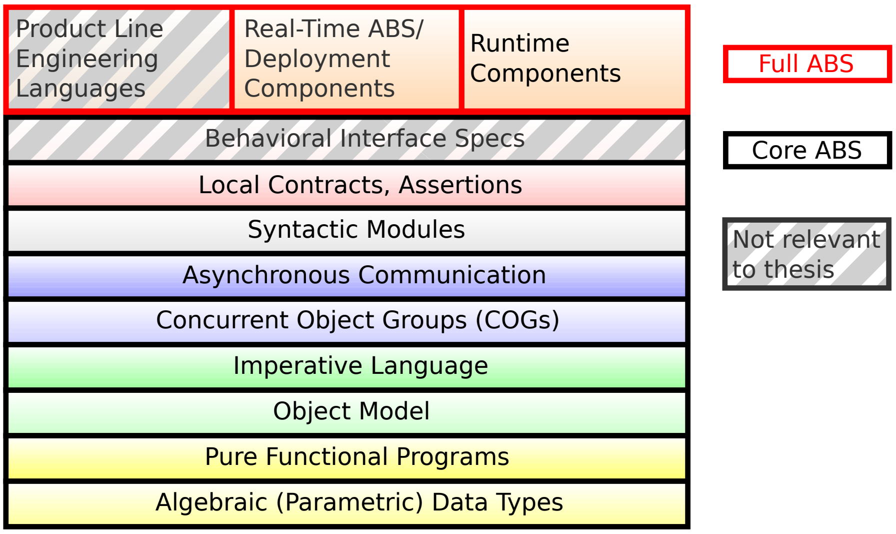

Product Line Engineering Languages
Real-Time ABS/ Deployment Components
Runtime Components
Full ABS
Behavioral Interface Specs
Core ABS
Local Contracts, Assertions
Syntactic Modules
Not relevant to thesis
Asynchronous Communication
Concurrent Object Groups (COGs)
Imperative Language
Object Model
Pure Functional Programs
Algebraic (Parametric) Data Types

Figure 2.1.: Layered architecture of ABS, image adapted from [20].

Algebraic Data Types (ADTs)

ABS provides a set of built-in data types which include among others `Bool`, `Int`, `Float` and `String`.

It also allows the user to define algebraic data types (ADTs) using the **`data`** keyword. Their constructors can take values of other types as parameters, see Listing 1, line 1. ADTs can also be parametric, that is, they can be defined with a type parameter which can be used in the type's constructors, see Listing 1, line 2.

In addition to the built-in types, some utility ADTs, like lists or maps (`List<T>`, `Map<K, V>`), and functions to work with them are provided by a standard library.

```
1  data IntTree = IntLeaf(Int) | IntNode(Tree, Tree);
2  data GenericTree<T> = Leaf(T) | Node(Tree<T>, Tree<T>);
```

Listing 1: Example of user-defined type definition for tree structures.

Functions

Functions in ABS can take an arbitrary number of arguments and their body is an expression from which their resulting value is computed upon invocation. ABS's functions are pure, which means that they can not have any side-effects on the model's state and will thus always return the same result given the same parameters[1]. Just like ADTs, functions can also be parametric to allow defining parameters of variable type.

Listing 2 gives two examples for function definitions and also showcases the destructuring of values via pattern matching using the `case` expression. This is a common pattern when working with ADTs.

```
def Int increment(Int x) = x + 1;

def Int countLeaves<T>(Tree<T> tree) =
  case tree {
    Leaf(x) => 1;
    Node(left, right) => countLeaves(left) + countLeaves(right);
  };
```

Listing 2: Examples of function definitions. The second function is parametric and makes use of the `case` expression.

As mentioned before, there are also second-order functions in ABS, which are called partially defined functions. These can take first-order functions as parameters and are useful for defining helper functions commonly used in functional languages, like `map` or `filter`. The example in Listing 3 defines a `filter` function on the predefined list type and demonstrates its use by passing an *anonymous function* as a parameter.

[1] The `random(Int)` function, which returns a random integer is an exception to this rule.

```
def List<T> myFilter<T>(fun)(List<T> lst) =
  case lst {
    Nil => Nil;
    Cons(head, tail) =>
      if (fun(head)) then
        Cons(head, myFilter(tail))
      else
        myFilter(tail);
  };

def List<Int> filterTheAnswer(List<Int> lst) =
  myFilter((Int x) => x == 42)(lst);
```

Listing 3: Example of a partially defined function and its application using an anonymous function definition.

2.2.3. Object Model and Imperative Language

In addition to ADTs, ABS allows also defining interfaces similar to Java which can be implemented by classes. However, classes do not define a type, therefore references to objects can only be typed by an interface of their class.

Interfaces

Interfaces can declare signatures of methods, which includes their name, return type and parameters. Methods can have the special empty return type `Unit` to indicate that they do not return a value. Listing 4 gives an example of an interface definition.

```
interface Printer {
  Bool preparePaper();
  Unit write(String text);
  Unit finalizePrint();
}
```

Listing 4: An example of an interface definition with 3 methods.

Classes

Class definitions in ABS contain field and method definitions[2]. They can implement multiple interfaces and must provide definitions for all of their method declarations. However, classes can not inherit from each other.

In contrast to functions, which are defined by expressions, methods are defined using an imperative language consisting of blocks of statements and control flow constructs. However, within expressions, the imperative language can make full use of all features of the functional layer.

See Listing 5 for an example of a class definition implementing the interface from Listing 4. The listing also includes a so-called *main block*, to demonstrate the instantiation of an object from the class definition. A *main block* is a program in the imperative language that is run upon executing a model.

[2] Fields can also only be declared and initialized later during instantiation of a class, but we do not make much use of this feature.

```
class HPOfficeJet implements Printer {
  Int sheetsOfPaper = 42;

  Bool preparePaper() {
    return sheetsOfPaper > 0;
  }

  Unit write(String text) {
    println(text);
  }

  Unit finalizePrint() {
    sheetsOfPaper = sheetsOfPaper - 1;
  }
}

// main block
{
  Printer p = new HPOfficeJet();
}
```

Listing 5: An example of a class implementing the interface of Listing 4 and a *main block* instantiating the class.

Modules

ABS requires source code to be organized in modules with identifying names so that the contents of another module can be imported by its name. See Listing 6 for an example.

A module can contain all of the aforementioned ABS language features, as illustrated in Figure 2.2. However, there can only be one main block per module. If there are multiple modules with main blocks, one needs to be selected when executing the model.

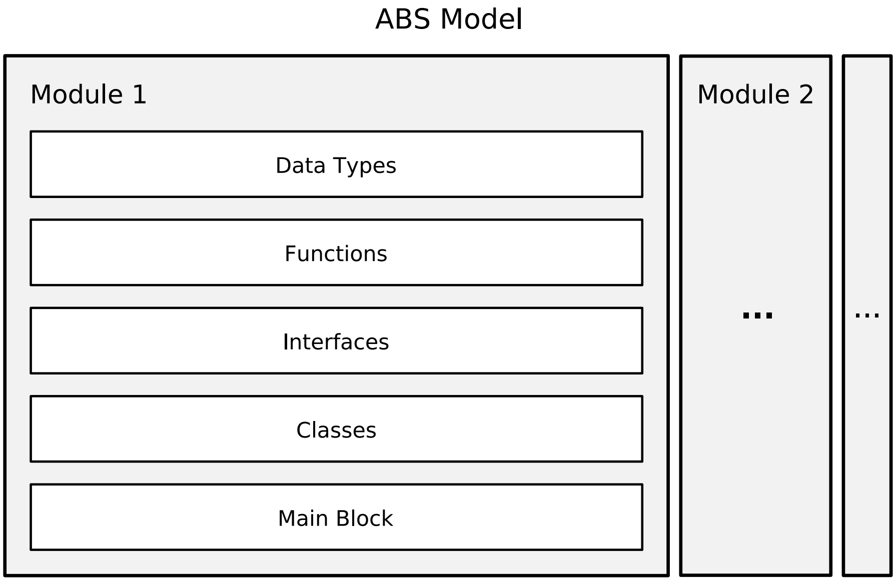


Figure 2.2.: Illustration of the structure of an ABS model specification.

```
module Printers; // module declaration

export Printer, HPOfficeJet;

interface Printer {
  // ...
}

class HPOfficeJet implements Printer {
  // ...
}

//...

module Main; // This can also be placed in another file

import * from Printers;

{
    Printer p = new HPOfficeJet();
    // ...
}
```

Listing 6: Example usage of the module system. In this code snippet, modules `Printers` and `Main` are created and the contents exported by `Printers` imported into `Main`.

2.2.4. Concurrency Model

Section 2.1 already hints at the concurrency model of ABS since the particular flavor of distributed systems ABS can describe is based on the same principles. This section now aims to give a more detailed account of ABS' concurrency model.

The concurrency model of ABS has been formalized in [27]. This supports its goal to ease the development of automated analysis and verification software. In ABS, so-called *Concurrent Object Groups* (COGs) are the language's source of concurrency. Each COG has a single thread of control and all COGs execute in parallel. Within a COG, ABS differs from main-stream languages like Java or C most prominently in the fact, that there is no preemptive scheduling. This means, that the execution of a method can not be interrupted at any moment to execute another method. Instead, ABS employs cooperative scheduling, which allows the programmer to release control by explicit statements at specific points or not at all.

Since it eliminates data races, cooperative scheduling makes it much easier for the programmer, but also for automated tools like the one developed in this thesis, to reason about a program with concurrently executing elements.

The following subsections now elaborate on the mentioned components of the concurrency model in more detail.

Concurrent Object Groups

By default, each object instantiated by the **`new`** expression and the main method have their own COG. An object can, however, be instantiated as part of the same COG as its creator when using the special **`new local`** expression.

Each object of a COG is associated with a pool of possible method activations. Initially, the pool is empty and the single thread of control executes an optional `init` block which can be defined for a class. As soon as a method is called asynchronously on a member of a COG, its activation is added to the COG's pool.

If the COG's thread of control is idle, one activation can be selected from the pool and executed on the single thread until it finishes or suspends execution due to cooperative scheduling. In the latter case, its reactivation at the suspension point is put back into the pool.

Scheduling Functions

From the above description of COGs, the question remains, how a method activation is selected from a pool. The actual definition of this selection method is left open by the ABS formalization [27], however, it is possible for the programmer to provide an implementation, called a *User-defined Scheduling Function*. *User-defined Scheduling Functions* are a part of Real-Time ABS and have been introduced by Bjørk et al. in [5].

A *User-defined Scheduling Function*, in short, a scheduling function, is defined just like any other function in ABS, but its signature must follow these restrictions:

1. its return type must be of the type `Process`
2. its first argument, we call it `queue`, must be typed `List<Process>`

The intention behind this signature is that the scheduling function takes the current pool of possible method activations as input, decides which activation to execute next by a custom algorithm and then outputs its choice. Please note, that scheduling functions are therefore called only with non-empty pools.

To make use of a scheduling function, classes can be annotated with its name (see Listing 7) so that all COGs resulting from instantiating the class must use it to select the next possible method activation.

```
def Process scheduler(List<Process> queue) = head(queue);

[Scheduler: scheduler(queue)]
class HPOfficeJet implements Printer {
  // ...
}
```

Listing 7: Definition and use of a *User-defined Scheduling Function* `scheduler` on a class `HPOfficeJet`. It always selects the first method activation from the pool of possible activations. The parameter passed in the annotation must be named `queue`.

Obviously, more complicated scheduling functions need some information about the possible activations and perhaps the object context they are executed in to make a decision. Firstly, the `Process` objects within the pool provide data about a method's name and invocation time. Secondly, the values of the annotated class's fields can be passed as additional parameters. See Listing 8 for an example.

```
def Process scheduler(List<Process> queue, Bool preferFoo) =
  case (queue) {
    Cons(p, Nil) => p;
    Cons(p, ps) =>
      if (preferFoo && method(p) == "foo") then
        p
      else
        scheduler(ps, preferFoo);
  };

[Scheduler: scheduler(queue, preferFoo)]
class C implements I {
  Bool preferFoo = True;

  Unit foo() { /* ... */ }
  Unit bar() { /* ... */ }
}
```

Listing 8: Example of a scheduling function which tries to always execute the method `foo`, given its activation is in the current pool and its object set the boolean flag `preferFoo`.

Calls

In ABS, the fields of an object are not directly accessible by other objects. Thus, the only means of communication between objects are synchronous and asynchronous method calls.

Asynchronous Calls An asynchronous call is an expression of the form

```
o!m(p0, p1, ..., pN)
```

where

- `o` is a pure expression evaluating to an object,
- `m` is the identifier of a method of one of the object's interfaces
- `p0`, `p1`, ..., `pN` are pure expressions typed in the same way as the `N` parameters of method `m`.

After an asynchronous call, execution of the calling method immediately continues. Instead of directly executing m, a corresponding method activation is added to the pool of the COG of o. Thus, the called method will only be executed when it gets scheduled for execution in this COG.

The call expression resolves to a future value of type Fut<T>, where T is the return type of the called method. See also the subsection regarding ABS's future values below.

Synchronous Calls A synchronous call is an expression of the form

```
o.m(p0, p1, ..., pN)
```

with the same meaning for the symbols o, m, p0, p1, ..., pN as above.

Depending on whether o is in the same COG as the object of the calling method, a synchronous call has slightly different effects:

Case: o is in the same COG In this case, execution directly progresses with the called method. After the called method finishes, it continues with the remaining statements of the calling method, where the call expression is resolved to the result returned by the called method.

Case: o is in a different COG Here, the synchronous call is equivalent to an asynchronous call and a suspension of execution of the own COG until the call is resolved.

Therefore, the synchronous call can be equivalently rewritten as an expression result with the following block of statements prepended:

```
Fut<T> f = o!m(p1, p2, ..., pN);
T result = f.get;
```

T is the return type of m. For an explanation of the get expression and Fut type, see the next subsection.

Futures

In ABS, as explained above, asynchronous calls return future values, see also Section 2.1.2. Those values are of the parametric type `Fut<T>`, where `T` is the return type of the method from which the future has been created.

The future value can be treated like any other data value, however, it provides two additional uses:

1. it can be used for cooperative scheduling to suspend execution until the asynchronous method execution it refers to finishes. This can be achieved using the Await Statement or Await Expression. For more information on those, you may consult the next section.
2. it can be used to retrieve the result of the asynchronous computation using a *Get Expression*.

   A Get Expression is of the form `f.get`, where `f` is a pure expression which evaluates to a future value. When a Get Expression is evaluated, it blocks all execution in its COG until the asynchronous computation finishes and its result is available. It then resolves to this result.

See Listing 9 for a demonstration of both use cases.

```
I p = new C();

Fut<Int> f = p!foo(); // Make asynchronous call

// resume execution...

// As soon as the result of foo() is needed, suspend execution
// until it is available.
await f?;

// Retrieve the result using a Get Expression
Int result = f.get;
println(toString(result));
```

Listing 9: Typical usage example of a future value.

Cooperative Scheduling Elements

The following statements and expressions can be used by the programmer to deliberately suspend execution, usually until a certain condition is met:

Await Statement The await statement suspends the execution of a method until a certain guard condition is met:

$$\boxed{\text{AwaitStmt}} \quad ::= \quad \texttt{await}\ \boxed{\text{Guard}};$$

As soon as the guard condition is met, the execution *can* be reactivated. The guard condition expression can be any pure expression, the identifier of a future or a conjunction of guard condition expressions. If the condition expression is a future identifier, then the guard is fulfilled, as soon as the computation referenced by the future has finished execution:

$$\boxed{\text{Guard}} \quad ::= \quad \boxed{\text{pure expression}} \mid \boxed{\text{future identifier}}? \mid \boxed{\text{Guard}} \,\&\, \boxed{\text{Guard}}$$

AwaitExp The await expression **`await`** `<Asynchronous Call>` is equivalent to the expression `x` after executing the following statements:

```
Fut<T> f = <Asynchronous Call>;
await f?;
T x = f.get;
```

Unconditional Suspend The **`suspend`** statement suspends the execution of the current method without any required condition for it to be resumed.

2.2.5. Model Simulation

The ABS tooling provides multiple so-called *backends* for translating an ABS model into an executable (programming) language so that its execution can be simulated by compiling and running the resulting code.

At the time of writing this thesis, there are backends for producing code in the languages Java, Haskell, Erlang and the rewriting logic Maude in different stages of completeness.

One of the goals of this thesis is the dynamic enforcement of behavior encoded by a protocol onto a simulated ABS model, which requires the use of scheduling functions (see Section 2.2.4). Right now, only the Erlang and Maude backends support scheduling functions.

Because some changes to future types and scheduling functions are applied to ABS during this thesis, they need to be reflected on the backend implementations, but due to the time constraints applying to this thesis we focus only on the Erlang backend.

However, the necessary changes to the Erlang backend are minor and do not require knowledge about the Erlang language of the reader.

2.3. Session Types

As pointed out in Section 2.1, distributed systems as a user application require the system's components to communicate in a structured fashion. Such a ”structure” can encompass not only the nature of interactions, but also their order, as well as repeated or alternative interaction sequences. A closed unit of such a communication is called a *session* [21].

If an implementation of a distributed system does not adhere to its intended communication structure, this will likely cause failures in the user application. Thus, a formal specification language for a session is desirable, since it allows the analysis of a distributed system in regard to a specification and may also aid its systematic development. *Multiparty session types* [21] are such a specification language. Originally, multiparty session types were intended to specify sessions in distributed systems with channel-based communication, as produced by the π-calculus. However, Kamburjan et al. [30] extended the session type notation so that they are suited for the concurrency model of ABS. Therefore, we make use of their approach on session types in this thesis.

2.3.1. Global Session Types

We call a session type specification language of the behavior of all objects of a session a *global session type*, since it operates from a global point of view. In our concurrency model of active objects, we want global types to capture the following aspects of communication and cooperative scheduling:

1. What kind of messages do active objects communicate between each other, i. e. which methods are called by whom on what object?

2. How are these interactions ordered?
3. What alternative interaction sequences are allowed?
4. Which parts of the interaction can be repeated?
5. At what point in the session has a method execution to be finished?
6. When are method executions suspended to wait for others?
7. When does an object try to retrieve the result of a call?

Aspects 1-4 allow reasoning about basic properties of structured communication, whereas aspects 5-7 cover details of the concurrency model of active objects. These are also relevant to communication, for example, the result of a call may only be retrieved, after its execution finished.

Of course, even more properties can be specified using session types, e. g. Kamburjan et al. [29] extended them to capture semantics of method calls by allowing to annotate calls with pre- and postconditions on their object's state. We also incorporate this idea on a smaller scale into our session type language at a later point in this thesis, see Section 3.7.

Syntax

From the need to specify aspects 1-7, the following syntax for a global session type, as introduced by Kamburjan et al. [30], arises:

$$
\begin{array}{lll}
\boxed{\mathrm{G}} & ::= & 0 \xrightarrow{f} p\colon m \;\mid\; \boxed{\mathrm{G}}.\boxed{\mathrm{g}} \\
\boxed{\mathrm{g}} & ::= & p \xrightarrow{f} q\colon m \;\mid\; p \downarrow f[(C)] \;\mid\; p \uparrow f[(C)] \;\mid\; \mathit{Rel}(p, f) \;\mid\; \mathit{Skip} \;\mid\; p\{\boxed{\mathrm{g}_i}\}_{i\in I} \;\mid\; (\boxed{\mathrm{g}})^*
\end{array}
$$

The symbols p and q range over identifiers of active objects, m ranges over method identifiers, f over futures, and C over ABS ADT constructor names. As evident by the above grammar, a session type can be a concatenation ($G.G'$) of other session types. If a part of a session type is optional, but its presence or absence is of no consequence to a statement, we write the optional part in square brackets, e. g. $p \downarrow f\;[(C)]$. Please note, that contrary to the session types of [30] our types do not encompass an **end** type which denotes termination. In our thesis, termination is implied by the syntactical end of a session type instead.

We denote the set of all global session types by $\mathcal{G}$ and its elements typically by lowercase g for non-concatenated and uppercase G for concatenated types. If a type g is part of a concatenated type G, we say g is a *component* of G. We call all types which contain no further nested types *atomic* session types. The set of atomic global session types is

$$\begin{aligned}\mathcal{G}_{\mathit{atomic}} = \{0 \xrightarrow{f} p\colon m, p \xrightarrow{f} q\colon m, p \downarrow f, p \downarrow f\,(C)\,, p \uparrow f, p \uparrow f\,(C)\,, \mathit{Rel}(p,f), \mathit{Skip} \mid \\ f \in \mathcal{F} \land p, q \in \mathit{Actors} \land m \in \mathit{Methods} \land C \text{ is ABS ADT constuctor}\}\end{aligned}$$

Definition 2.3.1 (Scope)**.** Futures are *introduced* by the types $0 \xrightarrow{f} q\colon m$ and $p \xrightarrow{f} q\colon m$. If a future f is introduced in a concatenated type $G.(\cdot) \xrightarrow{f} q\colon m.G'$, then we call $(\cdot) \xrightarrow{f} q\colon m.G'$ the *scope* of f.

Informal Semantics

$\mathbf{0} \xrightarrow{\mathbf{f}} \mathbf{p}\colon \mathbf{m}$ denotes the start of a session and is always the first component of a complete and valid type. It signifies that method m of actor p is called and f is the future representing the resulting computation.

$\mathbf{p} \xrightarrow{\mathbf{f}} \mathbf{q}\colon \mathbf{m}$ symbolizes that actor p calls method m on an actor q. The future produced by this call is f. We say “p interacts with q by calling m”.

$\boldsymbol{p \downarrow f\,[(C)]}$ expresses that the actor p finishes the computation referenced by future f. The (C) part is optional and designates that the resulting value of the computation has been created by applying the data constructor C. We say “p resolves f (with C)”.

$\boldsymbol{p \uparrow f\,[(C)]}$ conveys that actor p “fetches” the result of the computation represented by future f. Similar to the above type, (C) is optional and declares that the result has been constructed with constructor C.

$\boldsymbol{\mathit{Rel}(p,f)}$ denotes that the future currently active on actor p suspends its execution until future f has been resolved. During this suspension, other executions may be activated. We say “p releases control until f is resolved.” Please note, that the fetching type $p \uparrow f\,[(C)]$ above does not imply a release of control, even if the result of the fetched future f has not been computed yet. This is because we will be applying these types to ABS models where reading the result of an unfinished future blocks execution of the whole COG until f has finished.

Skip signifies no action.

$\boldsymbol{p}\left\{(\boldsymbol{G_i})_{\boldsymbol{i\in I}}\right\}$ expresses that there are $|I|$ different branches the session can continue with and actor p chooses which branch is taken. Each G_i encodes one possible branch.

$(\boldsymbol{G})^*$ expresses that the part of the behavior specified in G may take place zero or more times.

Example 1: Mail Notifications

Imagine a mobile mail application which comes with a notification service that informs the user about new mail as soon as it is available. The following global session type specifies the notification service's communication protocol with the mail server and the system's user interface:

$$
\begin{aligned}
&0 \xrightarrow{f_0} \mathit{NotificationService}\colon \mathit{init}. && (1)\\
&(&& \\
&\quad \mathit{NotificationService} \xrightarrow{f_1} \mathit{MailServer}\colon \mathit{checkMail}. && (2)\\
&\quad \mathit{Rel}(\mathit{NotificationService}, f_1). && (3)\\
&\quad \mathit{MailServer}\left\{
\begin{array}{l}
\mathit{MailServer} \downarrow f_1\,(\texttt{NewMail}).\\
\quad \mathit{NotificationService} \uparrow f_1\,(\texttt{NewMail}).\\
\quad \mathit{NotificationService} \xrightarrow{f_2} \mathit{UI}\colon \mathit{popup}.\\
\quad \mathit{UI} \downarrow f_2,\\
\mathit{MailServer} \downarrow f_1\,(\texttt{NoMail}).\\
\quad \mathit{NotificationService} \uparrow f_1\,(\texttt{NoMail})
\end{array}
\right\}. && (4)\\
&)^*. && (5)\\
&\mathit{NotificationService} \downarrow f_0 &&
\end{aligned}
$$

After the notification service has been initialized (line 1) it queries the mail server whether there has new mail arrived yet (line 2) and waits for an answer (line 3).

The mail server can now deliver different answers by resolving the future f_1 which has been produced by the query with different constructors. It can either answer that there is new mail to be retrieved, as in the first case of line 4, or it can state that there is no new mail, as in the second case. After reading the answer, the service will render a notification in the UI to inform the user about new mail but only in the first case.

> Since the service needs to check for new mail regularly, the interaction with the mail server can repeat indefinitely (line 5).

Self-Containedness

As mentioned, session types can specify actions which can be repeated an unspecified number of times, or not take place at all. However, some actions can not be repeated in an ABS model. Others must be executed at least once for the remaining specification to be valid.

> **Example: Resolving in a loop**
> The following session type specifies, that future f gets resolved potentially multiple times. However, a computation can finish only once:
>
> $$\ldots (p \downarrow f)^* .q \uparrow f \ldots$$
>
> Also, it would have to be resolved at least once so that q can fetch the result in the second part of the type.

To avoid types which describe impossible systems, we require repeated types to be self-contained. Formalizing and checking the requirements of self-containedness is one concern of our *Configurable Session Type Validation*, see Section 3.2.

Formal Semantics

We only give a brief overview of the formal semantics of session types. An in-depth formalization of the semantics by E. Kamburjan can be found in [28] for an almost identical definition of session types. We apply these formal semantics when discussing the soundness of our verification system in Sections 3.4.5 and 3.5.8.

E. Kamburjan defines histories of so-called *communication events* which are produced by the execution of an ABS model. These events describe operations on futures, like a call, an activation of a method, the suspension of a computation, etc. For example, the following history expresses that a method m_1 of an object o_1 is being called, activated and resolved. Then the same happens for a method m_2 of an object o_2:

$$\begin{aligned} h_{\mathit{example}} = [&\mathsf{invEv}(o_0, o_1, f_1, m_1, e_1), \mathsf{invREv}(o_0, o_1, f_1, m_1, e_1), \mathsf{futEv}(o_1, f_1, m_1, e_1), \\ &\mathsf{invEv}(o_0, o_2, f_2, m_2, e_2), \mathsf{invREv}(o_0, o_2, f_2, m_2, e_2), \mathsf{futEv}(o_2, f_2, m_2, e_2)] \end{aligned}$$

If we filter out the events produced by a specific object o or future f from a history h, we say h is projected onto o or f. For this we write $h \upharpoonright o$ and $h \upharpoonright f$ respectively:

$$h_{\mathit{example}} \upharpoonright o_1 = [\mathsf{invEv}(o_0, o_1, f_1, m_1, e_1), \mathsf{invREv}(o_0, o_1, f_1, m_1, e_1), \mathsf{futEv}(o_1, f_1, m_1, e_1)]$$

The formal semantics of a session type G are defined as a regular expression $\tau(G)$ describing a language of communication event histories. For example:

$$\tau(0 \xrightarrow{f} o\colon m.o \downarrow m) = [\mathsf{invREv}(0, o, f, m, \varepsilon)] \circ [\mathsf{futEv}(o, f, m, \mathrm{null})]$$

A history h produced by executing an ABS model is *captured* by a global session type if there is some future-equivalent[3]permutation h' of h in its language. When comparing h against permutations, we ignore concrete parameters and return values encoded in a history. Also, we exclude permutations where the order of events has been changed from the perspective of individual objects. That is we require $h \upharpoonright o = h' \upharpoonright o$ for all objects o.

The formal semantics of local session types (see below) are analogously defined as regular expressions.

2.3.2. Object Local Session Types

A global type specifies a session between multiple collaborating parties of a distributed system. To statically verify that the behavior of a program implementing the system conforms to a global session type, Honda et al. developed a process called *Projection* which allows extracting a specification of the behavior of a single party from the global type [21]. This *local* session type could then be used to perform verification for every party one at a time.

In the context of a concurrency model using active objects, Kamburjan et al. differentiate two kinds of local types, *object local* session types and *method local* session types [30]. *Object local* types describe a session from the view of a single active object and are thus useful for deriving scheduling strategies for an object, see Section 3.5.

[3] A history h is future-equivalent to another history h' if there is a renaming of futures in h so that it is equal to h'.

Syntax

As with global types, we employ the syntax introduced by Kamburjan et al. in [30] for object local types:

$$
\begin{aligned}
\boxed{\mathrm{L}} &::= \boxed{\mathrm{l}}[.\boxed{\mathrm{L}}] \\
\boxed{\mathrm{l}} &::= 0?_f m \;\mid\; p?_f m \;\mid\; p!_f m \;\mid\; \text{Put}\, f[(C)] \;\mid\; \text{Get}\, f[(C)] \;\mid\; \text{React}\, f \\
&\quad\;\mid\; \textit{Await}(f, f') \;\mid\; \textit{Skip} \;\mid\; \oplus\{\boxed{\mathrm{L}_i}\}_{i \in I} \;\mid\; \&_f\{\boxed{\mathrm{L}_i}\}_{i \in I} \;\mid\; (\boxed{\mathrm{L}})^*
\end{aligned}
$$

We denote the set of all object local session types by $\mathcal{L}$, non-concatenated object local types by lowercase l and concatenated types by uppercase L.

Informal Semantics

In the following informal description of the semantics of an object local session type, we refer to the active object whose behavior is specified by the type by q.

$\boldsymbol{0?_f m}$ expresses that at the start of the session, the active object q to which this type belongs, is called on method m, producing future f.

$\boldsymbol{p?_f m}$ denotes that q is called on method m by actor p, producing future f.

$\boldsymbol{p!_f m}$ symbolizes that q calls method m on actor p, producing future f.

Put $\boldsymbol{f}$ $[(\boldsymbol{C})]$ signifies q finishes the computation referenced by future f and creates its return value using ADT constructor C. Again, the (C) part is optional.

Get $\boldsymbol{f}$ $[(\boldsymbol{C})]$ conveys that q "fetches" the result of the computation represented by future f, which was constructed using C. Specifying (C) is optional.

Await$(\boldsymbol{f}, \boldsymbol{f'})$ denotes that the future f, which refers to an active computation in q, suspends its execution until the future f' has been resolved.

React $\boldsymbol{f}$ expresses that a suspended future f of q resumes its execution.

Skip signifies no action.

$\boldsymbol{\oplus \{(L_i)_{i \in I}\}}$ expresses that there are $|I|$ different branches the session can continue with and a computation in q actively chooses which branch is taken. For example, the computation could call different actors in each branch or have different return values, which are read by another actor and influence its behavior. Each L_i encodes one possible branch.

$\boldsymbol{\&_f \{(L_i)_{i \in I}\}}$ again expresses that there are different branches in the session. However, which branch is taken is not decided by q but by the computation referenced by future f. We say "q *offers* choices $(L_i)_{i \in I}$ to f."

$\boldsymbol{(L)^*}$ denotes that the part of the behavior specified in L may take place zero or more times.

Example 2: Mail Notifications (Part 2)

Looking back at the mail application scenario from Example 1, the behavior of the mail server could be captured by an object local session type like this:

$$\begin{aligned}
&(\\
&\quad \mathit{NotificationService}?_{f_1}\mathit{checkMail}. && (1) \\
&\quad \oplus \left\{ \begin{matrix} \text{Put } f_1 \ (\texttt{NewMail}) \\ \text{Put } f_1 \ (\texttt{NoMail}) \end{matrix} \right\}. && (2) \\
&)^*
\end{aligned}$$

Please note that the type has no notion of the actions of actors other than the mail server, except for the times when they are directly communicating with it. This is the case here when the mail server receives a call from the notification service in line 1. Line 2 makes use of the Choice Type $\oplus \{\ldots\}$, since the mail server actively chooses how the protocol progresses. Correspondingly, the same section of a local type describing the notification service would need to use the Offer Type $\&_{f_1} \{\ldots\}$ instead since the service's behavior depends on the return value of f_1.

2.3.3. Method Local Session Types

Method local types describe the behavior of individual methods and are thus useful for checking ABS programs method by method against a session type specification. Whereas Kamburjan et al. also employed the existing syntax of object local session types for method local types [30], we are going to introduce a separate, slightly changed syntax.

Our motivation for this is to simplify the static verification process by…

1. …removing types that are not explicitly represented in the AST of a method. That is $0?_f m$ and $p?_f m$, which always correspond to the beginning of a method. The type React f is thus also removed since it signifies the completion of **await** statements, which are already described by the *Await*(f, f') type.
2. …having Offer Types $\&_f \{\dots\}$ directly label every branch with constructor names corresponding to the different return values of the choosing future f. If a method's behavior branches depending on a return value, the static verification process needs to know which branch corresponds to which constructor label. Having explicit labels eliminates the need to inspect the branches for Get $f\ (C)$ types from the static verification process.

Syntax

$$\begin{array}{rcl}
\boxed{\mathrm{M}} & ::= & \boxed{\mathrm{m}}[.\boxed{\mathrm{M}}] \\
\boxed{\mathrm{m}} & ::= & p!_f m \ \mid\ \mathrm{Put}\ f[(C)] \ \mid\ \mathrm{Get}\ f[(C)] \ \mid\ \mathit{Await}(f, f') \ \mid\ \mathit{Skip} \\
& & \mid\ \oplus\{\boxed{\mathrm{M}_i}\}_{i \in I} \ \mid\ \&_f\{C_i : \boxed{\mathrm{M}_i}\}_{i \in I} \ \mid\ (\boxed{\mathrm{M}})^*
\end{array}$$

We denote the set of all method local session types by $\mathcal{M}$, and its elements by uppercase M.

Since the informal semantics of method local types are the same as of object local session types, we just like to reference Section 3.4 here. It revisits the mail application scenario of Examples 1 and 2 to give an intuition of how a method local type directly relates to the AST of an ABS method.

3. Concept

The main objective of this thesis is to develop a tool which is able to verify that an ABS model complies with a protocol specification based on session types. For this purpose, we build upon the theoretical work of Kamburjan et al. in [30] and [28]. Kamburjan et al. approach this challenge in two steps:

1. It must be statically verified that the implementation of every method locally complies with the actions specified for it in the protocol.
2. Activations and reactivations of these methods must be scheduled in the specified order on the object level.

Moreover, we incorporate ideas from [29] to extend session types with postconditions which allow reasoning about the effects of method executions on an active object's state. We verify these postconditions at runtime.

Therefore, we establish the following conceptual goals to be achieved in this thesis:

Goal I. Implementation of a specification language for communication protocols in distributed systems based on session types.

Goal II. Static verification of method-local compliance of an ABS model with a session type specification.

Goal III. Dynamic enforcement of method activation order and safety properties as specified by a session type.

Goal IV. Verification of postconditions of interactions between actors at runtime.

To realize goal I, a parser has been implemented which can read a session type specification from text files. It needs to produce a representation of session types as a data structure suited to achieve an implementation of goals II to IV. The details of how the parser has been implemented are presented in Chapter 4 in Section 4.3.

Furthermore, since it is possible to create session type specifications which are semantically invalid, it is necessary to validate the specification provided by the user. Moreover, the *Projection* process needed for static verification requires additional information about each component of a protocol specification that is not provided by its syntactic form. For example, it is necessary to know exactly which futures are supposed to be active at every point of a session protocol. Both, validation of a session type and its annotation with additional information is achieved by a process we call *Configurable Session Type Validation*, see Section 3.2.

We approach the challenge of accomplishing goals II and III by solving them for each actor and method individually. This requires us to extract object local and method local session types from a global session type specification by *Projection*. We present our version of Projection in Section 3.3.

Details on how we achieved goals II to IV are explained in Sections 3.4, 3.5 and 3.7. Figure 3.1 illustrates the dependencies between the goals and the major conceptual steps of Configurable Session Type Validation and Projection.

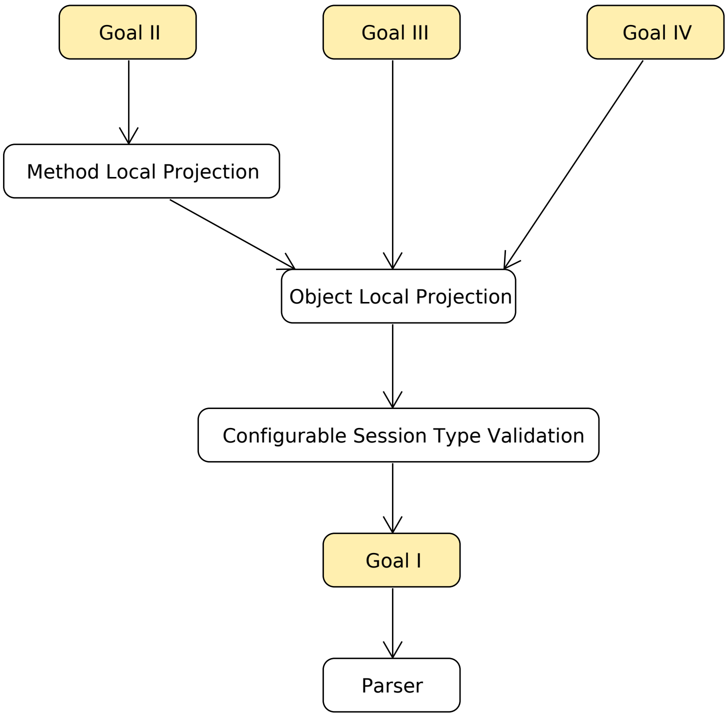


Figure 3.1.: Dependencies between major conceptual steps and thesis goals.

3.1. Tool Architecture

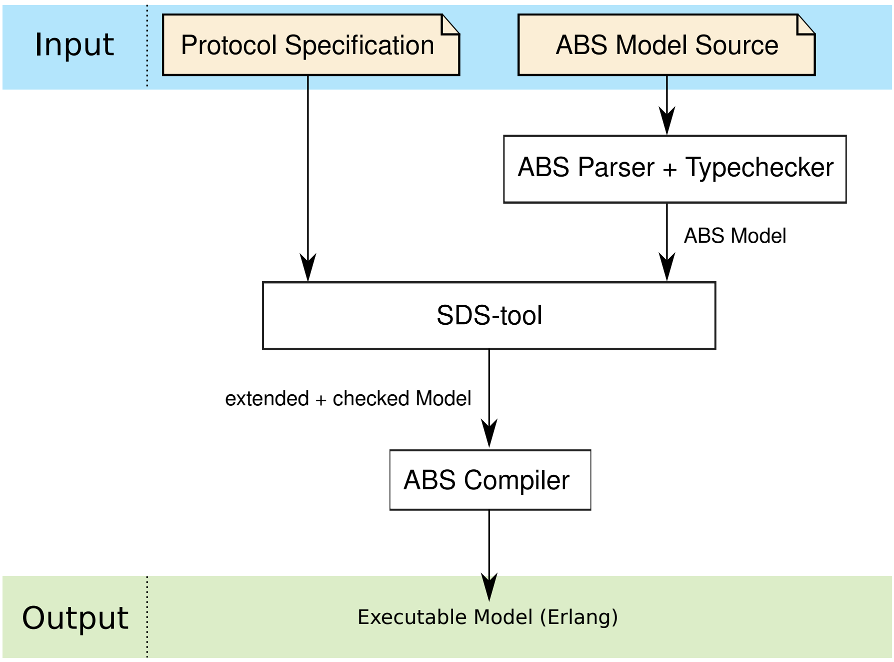


Figure 3.2.: Rough overview of the mechanics of the SDS-tool.

Figure 3.2 gives a rough overview of how the SDS-tool developed in this thesis operates. It takes protocol specifications as text files and the source code of an ABS model as input.

The source code is parsed and type-checked by the existing compiler for ABS [44], which produces an Abstract Syntax Tree (AST) of the model. Our tool then inspects this AST to verify statically that the model's methods locally conform to the communication correctness requirements imposed by the protocol specification. If the model passes the verification, it is then modified to carry out the dynamic enforcement goal and check postconditions at runtime. This modified AST is then passed on to the *abstools* backend compiler, which produces an executable Erlang program.

Figure 3.3 shows how the concepts introduced in this chapter make up the architecture of the tool in a more detailed way.

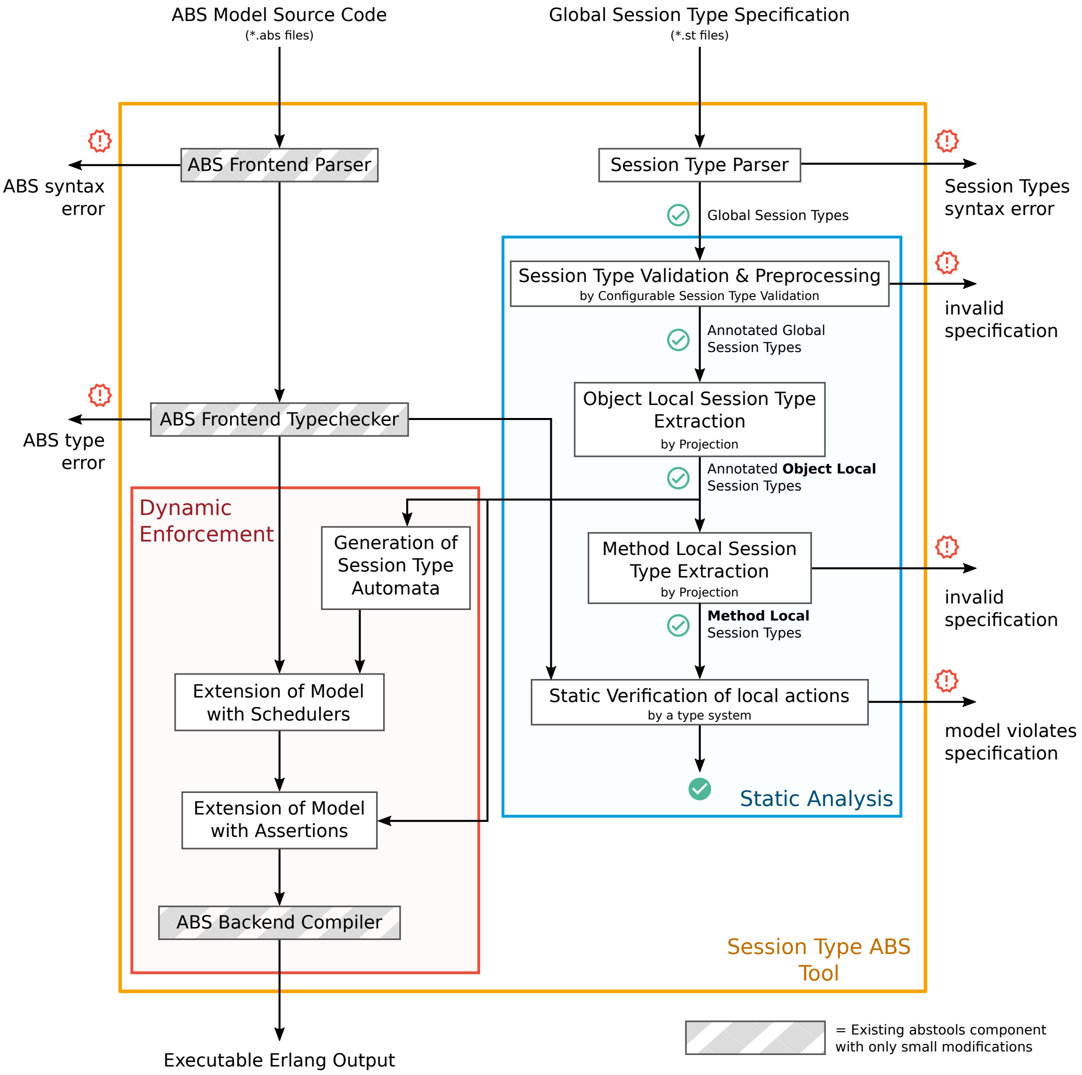

ABS Model Source Code
(*.abs files)
Global Session Type Specification
(*.st files)
ABS Frontend Parser
ABS syntax error
Session Type Parser
Session Types syntax error
Global Session Types
Session Type Validation & Preprocessing
by Configurable Session Type Validation
invalid specification
Annotated Global Session Types
Object Local Session Type Extraction
by Projection
ABS Frontend Typechecker
ABS type error
Annotated Object Local Session Types
Dynamic Enforcement
Generation of Session Type Automata
Method Local Session Type Extraction
by Projection
invalid specification
Method Local Session Types
Extension of Model with Schedulers
Static Verification of local actions
by a type system
model violates specification
Static Analysis
Extension of Model with Assertions
ABS Backend Compiler
Session Type ABS Tool
Executable Erlang Output
= Existing abstools component with only small modifications

Figure 3.3.: Detailed overview of the architecture of the developed tool.

3.2. Configurable Session Type Validation

As mentioned, we validate specifications and annotate them with additional information by executing a session type. This means the session type is split up into its atomic components which are treated as instructions and control structures of a program. This program is then executed on some state. A session type is considered invalid if the execution stops before processing the whole type. The pre- and poststates of applying an instruction carry the desired additional information necessary for Projection and are annotated onto the type.

Illustration

Consider the following session type:

$$0 \xrightarrow{f} p\colon m . p \xrightarrow{f'} q\colon m' . \underbrace{q \downarrow f'}_{\mathbf{A}} . \underbrace{p \uparrow f'}_{\mathbf{B}} . p \downarrow f$$

We can interpret this type as a protocol which specifies that actor p calls method m' on actor q, q then computes the result and p reads it.

To illustrate the analysis process, we now informally execute this type as a program on a state which records at what point in the protocol a future can be considered to be resolved. Therefore, we model state as a set of futures and start execution with the empty set as initial state. Execution is performed by a transition function $\rightsquigarrow$: *State* $\times$ $\mathcal{G}_{atomic}$ $\rightharpoonup$ *State* which is modeled so that a future is added to the state whenever a resolving type is encountered:

$$\underset{\text{initial state}}{\emptyset} \overset{0 \xrightarrow{f} p\colon m}{\rightsquigarrow} \emptyset \overset{p \xrightarrow{f'} q\colon m'}{\rightsquigarrow} \emptyset \overset{q \downarrow f'}{\rightsquigarrow} \{f'\} \overset{p \uparrow f'}{\rightsquigarrow} \{f'\} \overset{p \downarrow f}{\rightsquigarrow} \underset{\text{final state}}{\{f', f\}}$$

Suppose we implement our transition function $\rightsquigarrow$ as a partial function that does not execute a fetching type if the fetched future has not been resolved yet, i. e. it is not part of the state.

We can then easily see that the execution would not complete for the following type which specifies that p reads the result of calling m' before the call has been completed:

> $$0 \xrightarrow{f} p\colon m.p \xrightarrow{f'} q\colon m'.\underbrace{p \uparrow f'}_{\textbf{B}}.\underbrace{q \downarrow f'}_{\textbf{A}}.p \downarrow f$$
>
> Therefore, we could use this state model and transition function to validate that only resolved futures are read. It would also compute for every atomic component of a type which futures are supposed to be resolved at this point.

We call this concept of validating session types by executing them with a configurable transition function, configurable states, etc. the *Configurable Session Type Validation*. It has loosely been inspired by the *Configurable Program Verification* method which has been developed by Beyer, Henzinger and Théoduloz [4]. Validation of session types has originally [30, 29] been performed for the most part during Projection (see Section 3.3). The Configurable Session Type Validation thus separates the validation of a session type from the extraction of a localized specification. We argue that this modularization makes our tool easier to maintain. The previous theoretical works on Projection also retrieve contextual information (like resolved futures) by referencing and querying different parts of the specification whenever needed which is tricky to implement. Since contextual information is provided by the annotated pre- and poststates of each component of an executed session type, we have also separated out this concern from the Projection process.

To our knowledge, Configurable Session Type Validation is a new approach on accomplishing the above means. We occasionally refer to it in the remainder of this thesis by the abbreviation *Configurable Validation* or *CST Validation*.

3.2.1. Formalization: Configurable Session Type Analysis

A *configurable session type analysis* $\mathbb{D} = (D, \sigma_0, \rightsquigarrow, \mathit{merge}, \mathit{selfContained}, \mathit{closeScopes})$ consists of

- a set of states D
- an initial state $\sigma_0 \in D$
- a partial transition function which executes an atomic component of a session type on a given state, $\rightsquigarrow: D \times \mathcal{G}_{\mathit{atomic}} \rightharpoonup D$
- a partial function for merging branches, $\mathit{merge} : D \times D \rightharpoonup D$

- a predicate for checking whether a repetition is self-contained, $\mathit{selfContained} \subseteq D^2$.
- a partial function for closing scopes, $\mathit{closeScopes} : D \times D \rightharpoonup D$, since some analyses need to adjust their state if the scope of a future ends. It takes the prestate of a concatenated type and the poststate of its last component as input.

This structure of analyses has been derived from the *configurable program analyses* presented in [4]. The functions mentioned above are partial since them being undefined for some input type can be used by an analysis to signal that a type is invalid. Since there are multiple validity properties regarding session types, we decided to modularize Configurable Validation into multiple analyses ($\mathbb{P}$, $\mathbb{F}$, $\mathbb{A}$, $\mathbb{R}$), each responsible for a different family of validity properties. Moreover, if the session type specification language we use for our tool was to be extended in the future, validation of the extension could be achieved just by adding another analysis. The individual analyses and the validity properties they each inspect are explained in Sections 3.2.2 to 3.2.5. Self-containedness, in particular, is checked in analyses $\mathbb{A}$ and $\mathbb{R}$.

To avoid having to perform a Configurable Validation multiple times for every analysis, we also provide the Combined Analysis $\mathbb{C}$, which incorporates all the other analyses, see Section 3.2.6. The idea of modularizing and recombining analyses was inspired by the concept of *Composite Program Analyses* developed by Beyer et al. in [4].

Execution Function

The actual application of Configurable Validation with an analysis $\mathbb{D}$ on a concatenated session type has been formalized by the function $\mathit{execute}_{\mathbb{D}} : \mathcal{G} \rightharpoonup \mathcal{G}^{\mathbb{D}}$, see Figure 3.5. It takes a global session type as input and outputs the same session type but every component has been annotated with its pre- and poststate during execution. The function is only defined if execution does not stop prematurely, indicating an invalid type. We call a session type when successfully executed and annotated with an analysis $\mathbb{D}$, an *Analyzed Global Session Type* and denote it by $G^{\mathbb{D}}$. The syntax representing Analyzed Global Types is defined by the grammar in Figure 3.4. It simply annotates pre- and poststates in angle brackets behind a type, e. g. $p \downarrow f \left\langle \sigma_{pre}, \sigma_{post} \right\rangle$. For concatenated types $G^{\mathbb{D}}.g^{\mathbb{D}} \left\langle \sigma_{pre}, \sigma_{post} \right\rangle$ we simply write $G'^{\mathbb{D}} \left\langle \sigma_{pre}, \sigma_{post} \right\rangle$ with $G'^{\mathbb{D}} = G^{\mathbb{D}}.g^{\mathbb{D}}$ to access the annotation of its last component.

The partial function $\mathit{execute}_{\mathbb{D}}$ is a convenience wrapper around another function $\mathit{executeStep}_{\mathbb{D}}$. It selects the initial state σ_0 of the analysis as a starting state for the execution. Because

$$\begin{array}{rcl} \boxed{\mathrm{A}} & ::= & \boxed{\mathrm{a}}\langle \sigma_{pre}, \sigma_{post}\rangle \mid \boxed{\mathrm{A}}.\boxed{\mathrm{A}} \\ \boxed{\mathrm{a}} & ::= & G \mid p\{(\boxed{\mathrm{A}_i})_{i\in I}\} \mid (\boxed{\mathrm{A}})^* \end{array}$$

Figure 3.4.: Grammar of Analyzed Global Session Types. G denotes an atomic global session type. p is an actor, σ_{pre} and σ_{post} symbolize pre- and poststates.

all outmost scopes are closed when a concatenated type ends, *closeScopes* is applied to the result type. We continue by discussing *executeStep*$_{\mathbb{D}}$.

For atomic types g, see Case 3.1, *executeStep*$_{\mathbb{D}}$ annotates the input state σ as prestate and $\leadsto_{\mathbb{D}} (\sigma, g)$ as poststate, which is the application of the transition function of analysis $\mathbb{D}$ on the input state and g. It then recursively continues execution on the remainder of the input type using the poststate of g as next prestate.

Branching types are executed on each branch individually and the resulting states are merged into one state by applying *merge*$_{\mathbb{D}}$ pairwise, see Case 3.2. Execution then continues on the merged state. Since the end of every branch closes scopes, *closeScopes* is applied to each of them.

For repeated types, *executeStep*$_{\mathbb{D}}$ does continue on the poststate of the nested type, although behavior specified by repetitions is not required to take place at all. This is not an issue since they must be self-contained, see Section 2.3.1, which is checked. Keeping the poststate of the nested type allows analyses to retain the information that the repeated type is present in the protocol. Considering a repeated type closes scopes, *closeScopes* is applied, too.

$$
\begin{aligned}
&\mathit{execute}_{\mathbb{D}} : \mathcal{G} \rightharpoonup \mathcal{G}^{\mathbb{D}}\\
&\mathit{execute}_{\mathbb{D}}(G) = G^{\mathbb{D}}_{\text{result}} \left\langle \sigma_{\mathit{pre}}, \sigma'_{\mathit{post}} \right\rangle \text{ if } \quad G^{\mathbb{D}}_{\text{result}} \left\langle \sigma_{\mathit{pre}}, \sigma_{\mathit{post}} \right\rangle = \mathit{executeStep}_{\mathbb{D}}(\sigma_0, G)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \sigma'_{\mathit{post}} = \mathit{closeScopes}(\sigma, \sigma_{\mathit{post}})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{executeStep}_{\mathbb{D}} : D \times \mathcal{G} \rightharpoonup \mathcal{G}^{\mathbb{D}}\\
&\mathit{executeStep}_{\mathbb{D}}(\sigma, \emptyset) = \emptyset\\
&\mathit{executeStep}_{\mathbb{D}}(\sigma, g.G) =
\end{aligned}
$$

$$
\begin{cases}
g \left\langle \sigma, \rightsquigarrow(\sigma, g) \right\rangle .G^{\mathbb{D}} & \text{if} \quad g \in \mathcal{G}_{\mathit{atomic}} \qquad (3.1)\\
& \quad \wedge G^{\mathbb{D}} = \mathit{executeStep}_{\mathbb{D}}(\rightsquigarrow(\sigma, g), G)\\
p\left\{ \left(G^{\mathbb{D}}_i \left\langle \sigma, \sigma'_{i;\mathit{post}} \right\rangle \right)_{i \in I} \right\} \left\langle \sigma, \sigma_{\mathit{post}} \right\rangle .G^{\mathbb{D}} & \text{if} \quad g = p\left\{ (G_i)_{i \in I} \right\} \qquad (3.2)\\
& \quad \wedge \left(G^{\mathbb{D}}_i \left\langle \sigma, \sigma_{i;\mathit{post}} \right\rangle \right)_{i \in I}\\
& \qquad = \left(\mathit{executeStep}_{\mathbb{D}}(\sigma, G_i) \right)_{i \in I}\\
& \quad \wedge \forall i \in I.\, \sigma'_{i;\mathit{post}} = \mathit{closeScopes}(\sigma, \sigma_{i;\mathit{post}})\\
& \quad \wedge \sigma_{\mathit{post}} = \mathit{mergeSeq}_{\mathbb{D}}((\sigma'_{i;\mathit{post}})_{i \in I})\\
& \quad \wedge G^{\mathbb{D}} = \mathit{executeStep}_{\mathbb{D}}(\sigma_{\mathit{post}}, G)\\
\left(G^{\mathbb{D}}_{\circlearrowleft} \left\langle \sigma, \sigma'_{\mathit{post}} \right\rangle \right)^{*} \left\langle \sigma, \sigma'_{\mathit{post}} \right\rangle .G^{\mathbb{D}} & \text{if} \quad g = (G_{\circlearrowleft})^{*} \qquad (3.3)\\
& \quad \wedge G^{\mathbb{D}}_{\circlearrowleft} \left\langle \sigma, \sigma_{\mathit{post}} \right\rangle =\\
& \qquad = \mathit{executeStep}_{\mathbb{D}}(\sigma, G_{\circlearrowleft})\\
& \quad \wedge \sigma'_{\mathit{post}} = \mathit{closeScopes}(\sigma, \sigma_{\mathit{post}})\\
& \quad \wedge (\sigma, \sigma'_{\mathit{post}}) \in \mathit{selfContained}\\
& \quad \wedge G^{\mathbb{D}} = \mathit{executeStep}_{\mathbb{D}}(\sigma'_{\mathit{post}}, G)
\end{cases}
$$

$$
\mathit{mergeSeq}_{\mathbb{D}}((\sigma_1, \sigma_2, \ldots, \sigma_n)) =
\begin{cases}
\sigma & \text{if } n = 0\\
\sigma_1 & \text{if } n = 1\\
\mathit{mergeSeq}_{\mathbb{D}}((\mathit{merge}(\sigma_1, \sigma_2), \sigma_3, \ldots, \sigma_n)) & \text{otherwise}
\end{cases}
$$

Figure 3.5.: Formalization of Configurable Validation by executing a session type via the function *execute*$_{\mathbb{D}}$. $\mathbb{D}$ may be any configurable analysis with $\mathbb{D} = (D, \sigma_0, \rightsquigarrow, \mathit{merge}, \mathit{selfContained}, \mathit{closeScopes})$. We denote the end of a concatenated type by $\emptyset$.

3.2.2. Participants Analysis

We start with the simplest analysis, which we call the “Participants Analysis” $\mathbb{P}$. Its purpose is to determine which actors are participating in a global session type.

Given two global session types, this information can be used to decide whether an actor is participating in both. This is utilized in the developed SDS-tool to permit the specification of multiple protocols for an ABS model, as long as their participants do not intersect.

Execution of a type with this analysis does not perform any validation like the other analyses, it only collects data.

$$\mathbb{P} = (P, \sigma_{0;\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathit{merge}_{\mathbb{P}}, \mathit{selfContained}_{\mathbb{P}}, \mathit{closeScopes}_{\mathbb{P}})$$

States

Since this analysis shall gather actors, we model state as a set of actors:

$$P = 2^{\mathit{Actors}}$$

The initial state before executing any session type where no actor has yet been observed to participate in the protocol is therefore modeled as the empty set:

$$\sigma_{0;\mathbb{P}} = \emptyset$$

Transition Function

The transition function of $\mathbb{P}$ (see Figure 3.6) adds actors to a state when applied to any session type referring to actors, as in cases 3.4 and 3.5. Consequently, the state is not modified when encountering an actor which has already been added or when processing the *Skip* type, as in Case 3.6.

$$\rightsquigarrow_{\mathbb{P}} (\sigma, G) = \begin{cases} \sigma \cup \{p\} & \text{if} \quad G = 0 \xrightarrow{f} p\colon m \qquad (3.4) \\ & \quad \vee\, G = p \downarrow f\,[(C)] \\ & \quad \vee\, G = p \uparrow f\,[(C)] \\ & \quad \vee\, G = \mathit{Rel}(p, f) \\ \sigma \cup \{p, q\} & \text{if } G = p \xrightarrow{f} q\colon m \qquad (3.5) \\ \sigma & \text{if } G = \mathit{Skip} \qquad (3.6) \end{cases}$$

Figure 3.6.: Transition function of analysis $\mathbb{P}$.

Merging

To compile a list of all potentially participating actors, we need to consider every actor appearing in any branching of a protocol. When merging post-states of branches, we therefore compute their union:

$$\mathit{merge}_{\mathbb{P}}(\sigma_1, \sigma_2) = \sigma_1 \cup \sigma_2$$

Self-Containedness

The self-containedness of repetitions is not checked by this analysis. Thus, its $\mathit{selfContained}_{\mathbb{P}}$ predicate is defined to be fulfilled for all inputs

$$\mathit{selfContained}_{\mathbb{P}} = P \times P$$

Scope Closure

A scope being closed is not relevant to collecting participating actors, hence, the $\mathit{closeScopes}_{\mathbb{P}}$ function does not perform any changes or validation here:

$$\mathit{closeScopes}_{\mathbb{P}}(\sigma_{pre}, \sigma_{post}) = \sigma_{post}$$

3.2.3. Future Freshness Analysis

The “Future Freshness Analysis” $\mathbb{F}$ records created futures and what actors and methods they belong to. Furthermore, it validates that no future is introduced twice by a call. We also make sure a future can only be referenced within its scope. This way, it is unambiguous to which computation a future symbol refers to at all times.

$$\mathbb{F} = (F, \sigma_{0;\mathbb{F}}, \rightsquigarrow_{\mathbb{F}}, \mathit{merge}_{\mathbb{F}}, \mathit{selfContained}_{\mathbb{F}}, \mathit{closeScopes}_{\mathbb{F}})$$

States

Encountered future symbols are paired with the name of the method they are computing and their actor. We also keep a set of future symbols whose scope is currently accessible. Thus, state is modeled as a set of future-actor-method triples and a set of accessible futures:

$$F = 2^{\mathcal{F} \times \mathit{Actors} \times \mathit{Methods}} \times 2^{\mathcal{F}}$$

Since no future of a session has been created before performing the initializing action $0 \xrightarrow{f} p \colon m$, we define the initial state as the pair of empty sets:

$$\sigma_{0;\mathbb{F}} = (\emptyset, \emptyset)$$

Transition Function

The transition function of this analysis (see Figure 3.7) has been defined so that it adds a future-actor-method triple to the state, as soon as an initialization or interaction type is encountered (Case 3.7). It also marks the newly introduced future as accessible by adding it to the second component. For all other types, the state stays unchanged (cases 3.8 and 3.9).

Furthermore, the side-conditions of Case 3.7 ensure that an interaction type is only accepted if the specified future has not been introduced yet, i. e. there is no future with that name whose scope is accessible. Also, all other atomic types carrying future symbols (Case 3.8) are only accepted, if the future’s scope is accessible. This requires that the future is part of the second component of the current state.

$$\rightsquigarrow_{\mathbb{F}} ((\sigma_1, \sigma_2), G) =$$

$$\begin{cases} (\sigma_1 \cup \{(f,p,m)\}, \sigma_2 \cup \{f\}) & \text{if } (G = 0 \xrightarrow{f} p\colon m \vee G = p \xrightarrow{f} q\colon m) \\ & \quad \wedge (f, \cdot, \cdot) \notin \sigma_1 & (3.7) \\ (\sigma_1, \sigma_2) & \text{if } (\quad G = p \downarrow f\,[(C)] \\ & \qquad \vee G = p \uparrow f\,[(C)] \\ & \qquad \vee G = \mathit{Rel}(p, f)) \\ & \quad \wedge f \in \sigma_2 & (3.8) \\ (\sigma_1, \sigma_2) & \text{if } G = \mathit{Skip} & (3.9) \end{cases}$$

Figure 3.7.: Transition function of analysis $\mathbb{F}$.

Merging

For branching types, we allow the introduction of the same future symbol in multiple branches since their scopes are separate. However, they must reference the same actor and method. We make use of this requirement during static verification, which needs to check whether branches are equivalent.

Since *closeScopes*$_{\mathbb{P}}$ eliminates all newly introduced futures from the second component of the state, the second components are identical, and we can just continue with one of them.

$$\mathit{merge}_{\mathbb{F}}((\sigma_1, \sigma_2), (\sigma_1', \sigma_2')) = (\sigma_1 \cup \sigma_1', \sigma_2)$$
$$\text{if } \nexists f, p, p', m, m'.\, \{(f,p,m), (f,p',m')\} \subseteq (\sigma_1 \cap \sigma_1') \wedge (p \neq p' \vee m \neq m')$$

Self-Containedness

Self-containedness is not checked by this analysis, thus the predicate is always fulfilled:

$$\mathit{selfContained}_{\mathbb{F}} = F \times F$$

Scope Closure

Whenever a concatenated type ends, all futures which are not in a surrounding scope are no longer accessible. Therefore, we need to remove all futures from the second state component which were not already accessible in the prestate:

$$\mathit{closeScopes}_F((\sigma_1, \sigma_2), (\sigma_1', \sigma_2')) = (\sigma_1', \sigma_2' \setminus \sigma_2)$$

3.2.4. Actor Activity Analysis

The purpose of this analysis is to keep track, at which point of a session which futures are active on which actors, which futures are suspended and when they need to be reactivated. It also validates that only actors with an active future can perform any actions.

$$\mathbb{A} = (A, \sigma_{0;\mathbb{A}}, \rightsquigarrow_{\mathbb{A}}, \mathit{merge}_{\mathbb{A}}, \mathit{selfContained}_{\mathbb{A}}, \mathit{closeScopes}_{\mathbb{A}})$$

States

For every actor, we capture whether it is currently inactive, active or suspended. Therefore, state is modeled as a function which maps each actor to its symbolic activity status, “*Active*”, “*Inactive*” or “*Suspended*”.

If an actor is active or suspended, we also record the futures which are suspended and map them to the foreign futures they are waiting on. Moreover, if an actor is active, we store the currently active future.

This results in the following set of states:

$$A = \mathit{Actors} \to \mathit{ActivityStatus}$$

where

$$\begin{aligned}\mathit{ActivityStatus} = \{&\mathit{Inactive}\}\\ &\cup \left\{(\mathit{Active}, f, \mathit{Waiting}) \mid f \in \mathcal{F} \wedge \mathit{Waiting} \subseteq \mathcal{F}^2\right\}\\ &\cup \left\{(\mathit{Suspended}, \mathit{Waiting}) \mid \mathit{Waiting} \subseteq \mathcal{F}^2\right\}\end{aligned}$$

Before initiating a protocol, no actor is active. Consequently, all actors are marked “*Inactive*” in the initial state:

$$\sigma_{0;\mathbb{A}} = \{(p, \mathit{Inactive}) \mid p \in \mathit{Actors}\}$$

Transition Function

The transition function of this analysis (see Figure 3.8) has been designed so that it collects information about the activation of participants and futures. It performs some validation steps related to future activation, too.

In general, an actor may only perform an action if it is currently active ($\sigma(p) = (\mathit{Active} \ldots)$). For calls (cases 3.10, 3.11 and 3.12), the callee state is always set to "*Active*". However, the side-conditions ensure that the callee has been inactive or suspended and the caller been active before that. Also, if the callee has been suspended, it is ensured that its record of suspended futures is preserved (Case 3.12).

When releasing control until another future f completes, as portrayed by Case 3.15, an actor's state is switched to inactive and its active future f added to the waitlist of the actor's suspended futures. However, we only allow releasing control if there is not already a suspended future of the actor waiting for f. Otherwise, it would be ambiguous which future to resume when f completes, see the following example:

> **Example: Releasing control twice on the same future**
>
> $$0 \xrightarrow{f} p\colon m_1.p \xrightarrow{f'} q\colon m_2.\mathit{Rel}(p, f').q \xrightarrow{f''} p\colon m_3.\underbrace{\mathit{Rel}(p, f')}_{\text{(I)}}.\underbrace{q \downarrow f'}_{\text{(II)}}\ldots$$
>
> When f' is resolved (II), it is unclear, whether f or f'' shall resume execution on p if we would allow actor p to release control on f' again at (I).

When the executed type specifies an actor to resolve its active future f_A, as in Case 3.13, the actor is first marked as inactive or suspended, depending on whether it still has suspended futures. We employ the helper function *releaseActor* to deal with this distinction. This results in the intermediate state σ'. Then σ' is again updated so that all futures suspended on f_A and their actors are reactivated. The Case 3.15 also carries a side-condition *dontReactivateActives* which makes sure that a resolving action can not be specified if there are futures suspended on the resolved future whose actors are currently active. This is necessary since resolving a future always implicitly resumes futures suspended on it. However, resuming a future is impossible while another one is still active in the same actor. See also the following example:

Example: Reactivating a suspended future on an active object

$$0 \xrightarrow{f} p\colon m_1.p \xrightarrow{f'} q\colon m_2.\mathit{Rel}(p,f').q \xrightarrow{f''} p\colon .\underbrace{q \downarrow f'}_{\text{(I)}} \ldots$$

At (I) future f'' is active on p. If this resolving action was allowed, f would be required to be reactivated at this point. This is impossible since f'' is currently active and an active object always has only one thread of control.

Finally, a *Skip* type does not affect the state, as implemented by Case 3.16.

$$\rightsquigarrow_{\mathbb{A}} (\sigma, G) =$$

$$\begin{cases}
\sigma[p := (\mathit{Active}, f, \emptyset)] & \text{if } G = 0 \xrightarrow{f} p\colon m \wedge \sigma(p) = \mathit{Inactive} \qquad (3.10)\\[4pt]
\sigma[q := (\mathit{Active}, f, \emptyset)] & \text{if} \begin{array}[t]{l} \quad G = p \xrightarrow{f} q\colon m \\ \wedge\, \sigma(p) = (\mathit{Active}, \cdot, \cdot) \\ \wedge\, \sigma(q) = \mathit{Inactive} \end{array} \qquad (3.11)\\[4pt]
\sigma[q := (\mathit{Active}, f, \mathit{Waiting})] & \text{if} \begin{array}[t]{l} \quad G = p \xrightarrow{f} q\colon m \\ \wedge\, \sigma(p) = (\mathit{Active}, \cdot, \cdot) \\ \wedge\, \sigma(q) = (\mathit{Suspended}, \mathit{Waiting}) \end{array} \qquad (3.12)\\[4pt]
\sigma' \left[\left\{\begin{array}{l} (q, (\mathit{Active}, f, \mathit{Waiting}' \setminus \{(f, f_A)\})) \\ \mid q \in \mathit{Actors} \\ \wedge\, \sigma'(q) = (\mathit{Suspended}, \mathit{Waiting}') \\ \wedge\, (f, f_A) \in \mathit{Waiting}' \end{array}\right\}\right] & \text{if} \begin{array}{l} \quad G = p \downarrow f_A\, [(C)] \\ \wedge\, \sigma(p) = (\mathit{Active}, f_A, \mathit{Waiting}) \\ \wedge\, \sigma' = \mathit{releaseActor}(\sigma, p, \mathit{Waiting}) \\ \wedge\, \mathit{dontReactivateActives}(\sigma, f_A) \end{array} \qquad (3.13)\\[4pt]
\sigma & \text{if} \begin{array}[t]{l} \quad G = p \uparrow f\, [(C)] \\ \wedge\, \sigma(p) = (\mathit{Active}, \cdot, \cdot) \end{array} \qquad (3.14)\\[4pt]
\sigma[p := (\mathit{Suspended}, \mathit{Waiting} \cup (f_A, f))] & \text{if} \begin{array}[t]{l} \quad G = \mathit{Rel}(p, f) \\ \wedge\, \sigma(p) = (\mathit{Active}, f_A, \mathit{Waiting}) \\ \wedge\, (\cdot, f) \notin \mathit{Waiting} \end{array} \qquad (3.15)\\[4pt]
\sigma & \text{if } G = \mathit{Skip} \qquad (3.16)
\end{cases}$$

where

$$\mathit{dontReactivateActives}(\sigma, f_A) = \nexists q.(\sigma(q) = (\mathit{Active}, \cdot, \mathit{Waiting}'') \wedge (\cdot, f_A) \in \mathit{Waiting}'')$$

$$\mathit{releaseActor}(\sigma, p, \mathit{Waiting}) = \begin{cases} \sigma[p := \mathit{Inactive}] & \text{if } \mathit{Waiting} = \emptyset \\ \sigma[p := (\mathit{Suspended}, \mathit{Waiting})] & \text{otherwise} \end{cases}$$

Figure 3.8.: Transition function of analysis $\mathbb{A}$.

Merging

We want branching types only to be valid if we can be certain for every actor and future whether it is active, inactive or suspended after the branching. That is why the $\mathit{merge}_{\mathbb{A}}$ operator requires the $\mathbb{A}$ post-states of every branch to be equal:

$$\mathit{merge}_{\mathbb{A}}(\sigma_1, \sigma_2) = \sigma_1 \text{ if } \sigma_1 = \sigma_2$$

The following example demonstrates the kind of issues that would arise otherwise:

Example: Resolving a future in only one branch

$$0 \xrightarrow{f} p\colon m.\underbrace{p\begin{Bmatrix} p \downarrow f \\ \mathit{Skip} \end{Bmatrix}}_{\text{(I)}}.\underbrace{p \xrightarrow{f'} q\colon m}_{\text{(II)}}\ldots$$

If we were to allow a branching type like (I), then it is not ensured that actor p would be able to perform the call in (II). If p followed the first branch, it would be inactive at this point, if it followed the second one, it would be active.

We want to point out that restrictions such as these can potentially be lifted, however, it simplifies reasoning and the implementation of the developed tool. For example, we could keep track of state in a more fine-granular way, such that it is known, which exact parts of a state are ambiguous. Then a branching type with differing post-states for the branches could be acceptable if no later action depends on ambiguous parts of the state, e. g. there is no call like (II) in the above example. Actually, Beyer et al. implement this for their program analyses by using semi-lattices as abstract states [4].

Self-Containedness

Since the actions specified by a repetition type may be executed an unknown number of times or not at all, resulting in an ambiguous post-state, a similar reasoning as in the above section applies. Therefore, analysis $\mathbb{A}$ regards a repetition only as self-contained if the post-state of its inner type is equal to the state before executing the repetition type:

$$(\sigma_{\text{inner}}, \sigma_{\text{pre-state}}) \in \mathit{selfContained}_{\mathbb{A}} \Leftrightarrow \sigma_{\text{inner}} = \sigma_{\text{pre-state}}$$

Effectively, this requires futures to be active at the end of a repeated type if they were active at its beginning and to be suspended if they were suspended at its beginning. Please note that this also means that a future suspended at the beginning of a repeated type can not be reactivated within the type. This is because to be suspended again at the nested type's end requires to await a different future as before.

Scope Closure

Futures created in nested types (within branchings, repetitions) are always resolved within their nested type, as required by analysis $\mathbb{R}$, see Section 3.2.5. Therefore, they can not appear as a suspended or active future in its $\mathbb{A}$ post-state within the combined analysis.

Thus, no special actions are necessary upon closing scopes in the $\mathbb{A}$ analysis:

$$\mathit{closeScopes}(\sigma_{pre}, \sigma_{post}) = \sigma_{post}$$

3.2.5. Resolution Analysis

Execution of a session type in the "Resolution Analysis" $\mathbb{R}$ validates the correct usage of results of futures in a specification.

More precisely, it validates that a protocol can only specify fetching operations on futures which have completed their computation. It also validates that futures are resolved within their scope.

$$\mathbb{R} = (R, \sigma_{0;\mathbb{R}}, \rightsquigarrow_{\mathbb{R}}, \mathit{merge}_{\mathbb{R}}, \mathit{selfContained}_{\mathbb{R}}, \mathit{closeScopes}_{\mathbb{R}})$$

States

To check whether a future has been resolved yet, the analysis's states need to track a set of resolved futures. This set is extended every time a resolving type is processed.

Furthermore, it needs to record a set of futures which have been introduced in the current (nested) type so that it can validate that they have been resolved when reaching the end of the type.

Thus, states in $\mathbb{R}$ are modeled as pairs of sets of futures:

$$R = 2^{\mathcal{F}} \times 2^{\mathcal{F}}$$

Initially, no futures have been introduced or resolved, hence, the initial state is modeled as the pair of empty sets:

$$\sigma_{0;\mathbb{R}} = (\emptyset, \emptyset)$$

Transition Function

$\mathbb{R}$'s transition function (Figure 3.9) marks a future as resolved, as soon as it is specified in a resolving type. This is captured by Case 3.18. We do not need to check whether a future is resolved twice since the *Actor Activity Analysis* $\mathbb{A}$ implicitly verifies this by only allowing active futures to be resolved. Similarly, $\mathbb{A}$ also ensures that only active, non-resolved futures may be the target of releases of control, so we do not need to check for this either when processing release types (Case 3.20).

Case 3.19 validates that a fetching action may only be specified if the fetched future has been marked as resolved in the current state.

Case 3.17 records newly introduced future symbols in the second component of the state so that their resolution within their scope can be verified by *closeScopes*$_{\mathbb{R}}$.

$$\rightsquigarrow_{\mathbb{R}} ((\sigma_\downarrow, \sigma_*), G) = \left\{ \begin{array}{lll} (\sigma_\downarrow, \sigma_* \cup \{f\}) & \text{if} \quad G = 0 \xrightarrow{f} p\colon m & (3.17)\\ & \quad\; \lor G = p \xrightarrow{f} q\colon m & \\ (\sigma_\downarrow \cup \{f\}, \sigma_*) & \text{if } G = p \downarrow f\,[(C)] & (3.18)\\ (\sigma_\downarrow, \sigma_*) & \text{if} \quad G = p \uparrow f\,[(C)] & (3.19)\\ & \quad\; \land f \in \sigma_\downarrow & \\ (\sigma_\downarrow, \sigma_*) & \text{if } G = \mathit{Rel}(p, f) & (3.20)\\ (\sigma_\downarrow, \sigma_*) & \text{if } G = \mathit{Skip} & (3.21) \end{array} \right.$$

Figure 3.9.: Transition function of analysis $\mathbb{R}$.

Merging

After a branching type, information about the resolution status of futures is only needed for those futures which have been introduced before the branching type. This is because fetching actions may not operate on futures introduced in nested types. Thus, we compute the first part of the merged state from the intersection of recorded resolved futures of the post-states of all branches. We do not need to be concerned about the removal of futures which have been introduced before the branching type since…

(a) if a future f which has been introduced before the branching is resolved in a branch, it must be resolved in all branches. This is already being validated by the $\mathit{merge}_{\mathbb{A}}$ function of the "Actor Activity Analysis" $\mathbb{A}$. Thus, if $\mathit{merge}_{\mathbb{A}}$ is defined for the post-states of two branches, f would be marked as resolved in both post-states in analysis $\mathbb{R}$.

(b) no operation in $\rightsquigarrow_{\mathbb{R}}$ removes futures

Analogously, we can retrieve the set of futures introduced in the surrounding scope of the branching until now by computing the intersection of the second component of the post-states of branches.

$$\mathit{merge}_{\mathbb{R}}((\sigma_{\downarrow;1}, \sigma_{*;1}), (\sigma_{\downarrow;2}, \sigma_{*;2})) = (\sigma_{\downarrow;1} \cap \sigma_{\downarrow;2}, \sigma_{*;1} \cap \sigma_{*;1})$$

Self-Containedness

To validate that no future introduced outside of a repetition $(G)^*$ is resolved within G, the *selfContained*$_{\mathbb{R}}$ predicate of analysis $\mathbb{R}$ requires the futures marked as resolved in the post-state of G, executed on the pre-state of $(G)^*$ to be equal to the resolved futures in said pre-state.

In simpler terms, it checks that there is no type $p \downarrow f$ in $(G)^*$ where f is a future which has not been introduced in G but before the repetition.

$$\begin{gathered}((\sigma_{\downarrow;\text{pre-state}}, \sigma_{*;\text{pre-state}}), (\sigma_{\downarrow;\text{inner}}, \sigma_{*;\text{inner}})) \in \mathit{selfContained}_{\mathbb{R}} \\ :\Longleftrightarrow \\ \sigma_{\downarrow;\text{pre-state}} = \sigma_{\downarrow;\text{inner}}\end{gathered}$$

Scope Closure

Analysis $\mathbb{R}$ utilizes its *closeScopes*$_{\mathbb{R}}$ function to verify that newly introduced futures are resolved until the end of a (nested) type. This way, every future is resolved within its scope. Thus, *closeScopes*$_{\mathbb{R}}$ is only defined when all futures introduced in a (nested) type are a subset of those futures which have been resolved, as noted in the first state component. In the case that we are in a nested type, the futures introduced in it are isolated by computing the set difference with the futures already recorded in the prestate.

Consequently, we can then also delete all those futures from the second component.

$$\begin{aligned}\mathit{closeScopes}((\sigma_{\text{pre};\downarrow}, \sigma_{\text{pre};*}), (\sigma_{\downarrow}, \sigma_{*})) &= (\sigma_{\downarrow} \setminus F_{sub}, \sigma_{\text{pre};*}) \\ &\quad \text{if } F_{sub} = \sigma_{*} \setminus \sigma_{\text{pre};*} \wedge F_{sub} \subseteq \sigma_{\downarrow}\end{aligned}$$

3.2.6. Combined Analysis

The Combined Analysis allows to apply all of the above analyses in parallel to a type.

$$\mathbb{C} = (C, \sigma_{0;\mathbb{C}}, \rightsquigarrow_{\mathbb{C}}, \mathit{merge}_{\mathbb{C}}, \mathit{selfContained}_{\mathbb{C}}, \mathit{closeScopes}_{\mathbb{C}})$$

States

States in the Combined Analysis are modeled as cross product of the state sets of the other analyses so that a state in $\mathbb{C}$ can carry the information collected by all sub-analyses. Consequently, the initial state is defined as a tuple build from all initial states of the sub-analyses.

$$A = P \times F \times A \times R$$

$$\sigma_{0;\mathbb{C}} = (\sigma_{0;\mathbb{P}}, \sigma_{0;\mathbb{F}}, \sigma_{0;\mathbb{A}}, \sigma_{0;\mathbb{R}})$$

Operations

The transition function of the Combined Analysis applies the transition functions of the sub-analyses to each sub-analysis's state component individually, see Figure 3.10. The same concept applies to the other operations *merge*$_{\mathbb{C}}$, *selfContained*$_{\mathbb{C}}$ and *closeScopes*$_{\mathbb{C}}$, see Figures 3.11 to 3.13. In the case of the *selfContained*$_{\mathbb{C}}$ predicate operation, a state in the Combined Analysis fulfills the predicate if the states of all sub-analyses fulfill their implementation of the predicate.

$$
\begin{aligned}
&\rightsquigarrow_{\mathbb{C}} \left(\left(\sigma_{\mathbb{P}}, \sigma_{\mathbb{F}}, \sigma_{\mathbb{A}}, \sigma_{\mathbb{R}}\right), G\right) = (\\
&\qquad \rightsquigarrow_{\mathbb{P}} (\sigma_{\mathbb{P}}, G), \\
&\qquad \rightsquigarrow_{\mathbb{F}} (\sigma_{\mathbb{F}}, G), \\
&\qquad \rightsquigarrow_{\mathbb{A}} (\sigma_{\mathbb{A}}, G), \\
&\qquad \rightsquigarrow_{\mathbb{R}} (\sigma_{\mathbb{R}}, G) \\
&)
\end{aligned}
$$

Figure 3.10.: Transition function of the Combined Analysis $\mathbb{C}$.

$$
\begin{aligned}
&\mathit{merge}_{\mathbb{C}} \left(\left(\sigma_{1;\mathbb{P}}, \sigma_{1;\mathbb{F}}, \sigma_{1;\mathbb{A}}, \sigma_{1;\mathbb{R}}\right), \left(\sigma_{2;\mathbb{P}}, \sigma_{2;\mathbb{F}}, \sigma_{2;\mathbb{A}}, \sigma_{2;\mathbb{R}}\right)\right) = (\\
&\qquad \mathit{merge}_{\mathbb{P}}(\sigma_{1;\mathbb{P}}, \sigma_{2;\mathbb{P}}), \\
&\qquad \mathit{merge}_{\mathbb{F}}(\sigma_{1;\mathbb{F}}, \sigma_{2;\mathbb{F}}), \\
&\qquad \mathit{merge}_{\mathbb{A}}(\sigma_{1;\mathbb{A}}, \sigma_{2;\mathbb{A}}), \\
&\qquad \mathit{merge}_{\mathbb{R}}(\sigma_{1;\mathbb{R}}, \sigma_{2;\mathbb{R}}) \\
&)
\end{aligned}
$$

Figure 3.11.: Merge operator of the Combined Analysis $\mathbb{C}$.

$$
\begin{aligned}
&(\\
&\qquad (\sigma_{\text{inner};\mathbb{P}}, \sigma_{\text{inner};\mathbb{F}}, \sigma_{\text{inner};\mathbb{A}}, \sigma_{\text{inner};\mathbb{R}}), \\
&\qquad (\sigma_{\text{pre-state};\mathbb{P}}, \sigma_{\text{pre-state};\mathbb{F}}, \sigma_{\text{pre-state};\mathbb{A}}, \sigma_{\text{pre-state};\mathbb{R}}) \\
&) \in \mathit{selfContained}_{\mathbb{C}} \\
&\Leftrightarrow \bigwedge_{i \in \{\mathbb{P},\mathbb{F},\mathbb{A},\mathbb{R}\}} (\sigma_{\text{inner};i}, \sigma_{\text{pre-state};i}) \in \mathit{selfContained}_{i}
\end{aligned}
$$

Figure 3.12.: Self-Containedness check of the Combined Analysis $\mathbb{C}$.

$$
\begin{aligned}
&\mathit{closeScopes}_{\mathbb{C}}\left((\sigma_{1;\mathbb{P}}, \sigma_{1;\mathbb{F}}, \sigma_{1;\mathbb{A}}, \sigma_{1;\mathbb{R}}), (\sigma_{2;\mathbb{P}}, \sigma_{2;\mathbb{F}}, \sigma_{2;\mathbb{A}}, \sigma_{2;\mathbb{R}})\right) = (\\
&\quad \mathit{closeScopes}_{\mathbb{P}}(\sigma_{1;\mathbb{P}}, \sigma_{2;\mathbb{P}}),\\
&\quad \mathit{closeScopes}_{\mathbb{F}}(\sigma_{1;\mathbb{F}}, \sigma_{2;\mathbb{F}}),\\
&\quad \mathit{closeScopes}_{\mathbb{A}}(\sigma_{1;\mathbb{A}}, \sigma_{2;\mathbb{A}}),\\
&\quad \mathit{closeScopes}_{\mathbb{R}}(\sigma_{1;\mathbb{R}}, \sigma_{2;\mathbb{R}})\\
&)
\end{aligned}
$$

Figure 3.13.: Scope closing operation of the Combined Analysis $\mathbb{C}$.

3.3. Projection

Projection is the process of extracting a local session type specification of the behavior of a single party from a multiparty global session type [21]. In our case of a concurrency model of active objects, we also differentiate between projection on objects and projection on methods [30]. Projection thus allows us to implement dynamic enforcement (goal III) and static verification (goal II) on a per-object and per-method basis respectively.

$$
\begin{aligned}
\text{Global Type } G &\xrightarrow{\mathit{execute}_{\mathbb{C}}(G)} \text{Analyzed Global Type } G^{\mathbb{C}}\\
&\xrightarrow{\mathit{project}^{t}(G^{\mathbb{C}})} \text{Analyzed Object Local Type } L^{\mathbb{C}}\\
&\xrightarrow{\mathit{project}^{t}_{f}(L^{\mathbb{C}})} \text{Method Local Type } M
\end{aligned}
$$

3.3.1. Object Local Projection

Projection on active objects is a partial function which maps Analyzed Global Types of the Combined Analysis to *Analyzed Object Local Types* for a given target object t:

$$\mathit{project}^{t} : \mathcal{G}^{\mathbb{C}} \rightharpoonup \mathcal{L}^{\mathbb{C}}$$

Analyzed Object Local Types are defined analogously to Analyzed Global Types in Figure 3.14. They are object local session types annotated with the pre- and poststates of the global type they were projected from.

Although almost all validation of session types is performed by our CST Validation, there are a few exceptions which are validated throughout object and method projection. Hence, projection is a partial function which is not defined for invalid session types. Projection works by only keeping those components of a more global type which directly involve the target we are projecting on and translating them into a localized type syntax. The other components result in the *Skip* type. When projecting on objects, we also copy annotated state information since it is required to perform method projection and static analysis. Our projection functions are based on the ones developed by Kamburjan et al. in [30] but they have mostly been stripped of validation steps since those are handled by the CST Validation. Also, they draw from state information.

For calls, we keep only those where the target object is either the caller or the callee, see cases 3.22 and 3.23.

A resolving type $p \downarrow f\,[(C)]$ produces the local Put $f\,[(C)]$ type, if the target t of the projection is object p resolving future f, see case 3.24. It can also result in the React f' type if our target object t does not resolve f but there is a future f' of t which has been waiting for the completion of f (case 3.25).

A fetching type $p \uparrow f\,[(C)]$ is translated into the equivalent Get f type if the target object is performing the fetching action, see case 3.26.

Releases of control *Rel*(p, f) are projected to the object local counterpart *Await*(f_A, f), where f_A is the future that has been active on p and which should from now on await the completion of f. However, this only applies if we are projecting on $p = t$. Other actors do not need to implement behavior for this type, therefore, we project to *Skip* for them. See case 3.27.

We keep repetitions $\left(G^{\mathbb{C}}\right)^*$ and project their content $G^{\mathbb{C}}$ (case 3.28). Yet, if the target object is not participating in the nested type of a repetition, that is, projecting the nested type produces *Skip*, we remove the repetition brackets and only keep the *Skip*.

Branchings $p\,\{\ldots\}$ are projected branch-wise to the Choice-Type $\oplus\,\{\ldots\}$ if the target is the object p choosing the branch to take (case 3.29) and to the Offer-Type $\&_f\,\{\ldots\}$ otherwise (case 3.30). This is also the one case where object projection can be undefined if the choosing object p is not active, as revealed by the $\mathbb{A}$-analysis. We could also check for this during Configurable Validation (Section 3.2), but then we would need to make

$$
\begin{array}{lcl}
\boxed{\mathrm{B}} & ::= & \boxed{\mathrm{b}}\langle \sigma_{pre}, \sigma_{post} \rangle \mid \boxed{\mathrm{B}}.\boxed{\mathrm{B}} \\
\boxed{\mathrm{b}} & ::= & L \mid \&_f\{(\boxed{\mathrm{B}_i})_{i \in I}\} \mid \oplus\{(\boxed{\mathrm{B}_i})_{i \in I}\} \mid (\boxed{\mathrm{B}})^*
\end{array}
$$

Figure 3.14.: Grammar of Analyzed Object Local Session Types. L denotes an atomic object local session type. f is a future, σ_{pre} and σ_{post} symbolize pre- and poststates of analysis $\mathbb{C}$.

executeStep$_{\mathbb{D}}$ aware of the $\mathbb{A}$-analysis. However, we wanted to keep *executeStep*$_{\mathbb{D}}$ flexible and generic and thus not specialized on some analysis.

When projecting concatenated types, we project them component-wise and remove all intermediate resulting *Skip* types, see case 3.31. Not having to worry about encountering *Skip*'s in concatenated types simplifies further processing slightly.

A single *Skip* type is preserved by projection, see case 3.32.

$$\mathit{project}^t(0 \xrightarrow{f} p\colon m \,\langle \sigma_{pre}, \sigma_{post}\rangle) = \begin{cases} p?_f m \,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } t = p \\ \mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{otherwise} \end{cases} \tag{3.22}$$

$$\mathit{project}^t(p \xrightarrow{f} q\colon m \,\langle \sigma_{pre}, \sigma_{post}\rangle) = \begin{cases} q!_f m \,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } t = p \\ p?_f m \,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } t = q \\ \mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{otherwise} \end{cases} \tag{3.23}$$

$$\begin{aligned} &\mathit{project}^t(p \downarrow f\,[(C)]\,\langle \sigma_{pre}, \sigma_{post}\rangle) = \\ &\quad\begin{cases} \text{Put } f\,[(C)]\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } t = p \qquad\qquad\qquad\qquad\qquad (3.24) \\ \text{React } f'\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } \quad t \neq p \qquad\qquad\qquad\qquad\qquad (3.25) \\ & \quad \wedge\, \mathit{sub}_{\mathbb{A}}(\sigma_{pre})(p) = (\mathit{Suspended}, \mathit{Waiting}) \\ & \quad \wedge\, (f', f) \in \mathit{Waiting} \\ \mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{otherwise} \end{cases} \end{aligned}$$

$$\mathit{project}^t(p \uparrow f\,[(C)]\,\langle \sigma_{pre}, \sigma_{post}\rangle) = \begin{cases} \text{Get } f\,[(C)]\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } t = p \\ \mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{otherwise} \end{cases} \tag{3.26}$$

$$\mathit{project}^t(\mathit{Rel}(p, f)\,\langle \sigma_{pre}, \sigma_{post}\rangle) = \begin{cases} \mathit{Await}(f_A, f)\,\langle \sigma_{pre}, \sigma_{post}\rangle & \quad\text{if } t = p \\ & \wedge\, \mathit{sub}_{\mathbb{A}}(\sigma_{pre})(p) = (\mathit{Active}, f_A, \mathit{Waiting}) \\ \mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{otherwise} \end{cases} \tag{3.27}$$

$$\mathit{project}^t\big((G^{\mathbb{C}})^*\,\langle \sigma_{pre}, \sigma_{post}\rangle\big) = \begin{cases} (L^{\mathbb{C}})^*\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } \quad \mathit{project}^t(G^{\mathbb{C}}) = L^{\mathbb{C}} \\ & \quad \wedge\, L^{\mathbb{C}} \neq \mathit{Skip}\,\langle \cdot, \cdot\rangle \\ \mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{otherwise} \end{cases} \tag{3.28}$$

$$\begin{aligned} &\mathit{project}^t(p\,\big\{(A_i)_{i\in I}\big\}\,\langle \sigma_{pre}, \sigma_{post}\rangle) = \\ &\quad\begin{cases} \oplus\,\big\{(\mathit{project}^p(A_i))_{i\in I}\big\}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } t = p \qquad\qquad\qquad\qquad\qquad (3.29) \\ \&_f\,\big\{(\mathit{project}^t(A_i))_{i\in I}\big\}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if } \quad t \neq p \qquad\qquad\qquad\qquad\qquad (3.30) \\ & \quad \wedge\, \mathit{sub}_{\mathbb{A}}(\sigma_{pre})(p) = (\mathit{Active}, f, \mathit{Waiting}) \end{cases} \end{aligned}$$

$$\mathit{project}^t(A.A') = \begin{cases} L^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle & \text{if} \qquad \mathit{project}^t(A) = L^{\mathbb{C}}\,\langle \sigma_{pre}, \cdot\rangle \\ & \qquad \wedge\, \mathit{project}^t(A') = \mathit{Skip}\,\langle \cdot, \sigma_{post}\rangle \\ & \quad \vee \quad \mathit{project}^t(A) = \mathit{Skip}\,\langle \sigma_{pre}, \cdot\rangle \\ & \qquad \wedge\, \mathit{project}^t(A') = L^{\mathbb{C}}\,\langle \cdot, \sigma_{post}\rangle \\ \mathit{project}^t(A).\mathit{project}^t(A') & \text{otherwise} \end{cases} \tag{3.31}$$

$$\mathit{project}^t(\mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle) = \mathit{Skip}\,\langle \sigma_{pre}, \sigma_{post}\rangle \tag{3.32}$$

Figure 3.15.: Projection function for producing object local session types from global session types for a given actor $p \in \mathit{Actors}$

3.3.2. Method Local Projection

Projection on methods is a partial function mapping the Analyzed Object Local Type of an object t to a method local type for a target future f:

$$\mathit{project}^t_f : \mathcal{L}^{\mathbb{C}} \rightharpoonup \mathcal{M}$$

We are projecting for a particular future f instead of a method because there may be multiple calling actions specified for the same method in a session type. We allow projection results for futures which have been created from the same method to differ. However, our static analysis ensures that the method's implementation conforms to all of them. Method projection has been formalized in Figure 3.16.

Atomic components are copied from the object local session type without modification if the future we are projecting on is currently active (case 3.33). However, as explained in Section 2.3.3, receiving and reactivating types are *Skip*ped, see cases 3.34 and 3.35. Concatenated types are projected component-wise, see case 3.36.

For repetitions $(\ldots)^*$, we project on the nested type and remove the repetition brackets if the future we are projecting on is introduced within the repetition (case 3.37). This is because the method typed by the repetition is supposed to be called within the loop and does not implement it. However, if the future existed beforehand and projection of the nested type does not result in the *Skip* type, we keep the repetition with the projected type inside since the method then must implement a looping statement (case 3.38). As pointed out in Section 3.2.4, the projection of the nested type can only result in a type different from *Skip* if the future was active when entering the loop. Otherwise, we simply *Skip* the repetition, see case 3.39.

Choice-Types $\oplus\{\ldots\}$ are projected branch-wise. We *Skip* the type entirely if all branches result in a *Skip* type (case 3.41).

Offer-Types $\&_{f'}\{\ldots\}$ are the only types for which method projection can be undefined if an invalid type is detected. Here we differentiate between 3 cases for the future f we are projecting on:

(a) Either f is introduced in one branch…

(b) …or f does not participate in the branching and has the same behavior in all branches.

(c) …or f reads from the choosing future f' and changes behavior depending on the result.

In the simplest case (a), we just need to project the branch f is introduced in. See projection case 3.42.

In case (b) we project every branch on f and check whether they are all future-equivalent. If this case applies, we select one of them as projection result. This is carried out by the helper function *identicalBranches*, see projection case 3.43 and equation 3.49. Our version of future-equivalence for method local types is disclosed further below.

Otherwise, case (c) applies. The projection then needs to perform the following steps:

S1 project the type branch-wise and check whether f reads from f' in every branch, Also, each time the result must be built using a different constructor than in all other branches.

$$\&_{f'} \left\{ \begin{array}{c} \cdots\ \mathrm{Get}\ f' \left(C_1 \right) \cdots \\ \cdots\ \mathrm{Get}\ f' \left(C_2 \right) \cdots \\ \vdots \\ \cdots\ \mathrm{Get}\ f' \left(C_n \right) \cdots \end{array} \right\}$$

S2 if so, extract the part of each branch up until that point (marked in green).

S3 check whether these parts are equivalent for all branches. Otherwise, let projection fail (undefined result).

S4 put the equivalent part before the Offer-Type and remove it from the branch projections. Also, remove the fetching types Get f' (C_i) and put it in front of the Offer-Type without a constructor annotation. Instead, label each branch by the unique constructor.

$$\cdots\ \mathrm{Get}\ f'.\&_{f'} \left\{ \begin{array}{c} C_1 : \cdots \\ C_2 : \cdots \\ \vdots \\ C_3 : \cdots \end{array} \right\}$$

Step S1 is necessary to ascertain that case (c) really applies. Then, the computation referenced by f must behave the same for all branches up until the point f reads from f' (steps S2 and S3). This is because f does not have any information about which choice f' made before then. If the specification violates this requirement, it is invalid. Thus, we must abort projection. We do not check this validity property during CST Validation because it requires us to view the behavior of every future in isolation. Hence, method projection is more suited to this task.

During static verification, we have to check whether methods with Offer-Types have control structures (**case** statements) such that their behavior depends on the constructor of the fetched future f'. To make verification easier, we have to know which constructor belongs to which branch of the specification. Therefore we perform step S4. Since the behavior encoded in the identical parts of the branches needs to be implemented preceding the control structure within a method, we also move the identical parts and the fetching action in front of the Offer-Type. This way, the session type can be processed component-wise during verification without having to analyze Offer-Types.

All steps are performed as part of the function *extractBranchLabels* and its helper function *splitOnGet* and helper set *Splits*. The function *splitOnGet*, see equation 3.48, computes all possible ways of splitting a branch specification into a part preceding a fetching action of f' by f and a part following it. Now, the set *Splits*, see equation 3.47, filters these results so that only those splits remain where all branches share the same prefix before the fetching action. Finally, *extractBranchLabels* selects the split with the smallest prefix from *Splits*, see equation 3.46. However, this decision restricts the space of methods which conform to the protocol and are verifiable. The static verification expects an Offer-Type to directly be implemented as a control structure (**case**-statement). Imagine there are two split possibilities in *Splits* with prefixes p_1 and p_2, where p_2 is longer i. e. $p_2 = p_1.p'$. Both, methods implementing the behavior of p' before and inside the control structure are conforming to the type. Nonetheless, only the latter is verifiable by our technique since we are always choosing the smaller prefix. This could be improved by encoding every possible prefix in the type and trying to verify every possibility. On the other hand, we did not want to introduce even more complexity to the projection and verification.

Example 3: Mail Notifications – Method Projection

Once again, we revisit the mail application from Examples 1 and 2 to illustrate the projection of an Offer-Type onto a future.

The following is an object local type describing the behavior of the notification service.

For the sake of brevity, we leave out pre- and poststates:

$$
\begin{aligned}
&0?_{f_0}\mathit{init}.\\
&(\\
&\quad \mathit{MailServer}!_{f_1}\mathit{checkMail}.\\
&\quad \&_{f_1}\left\{\begin{array}{l}
\mathit{Await}(f_0, f_1).\\
\quad \text{React } f_0.\\
\quad \text{Get } f_1\ (\texttt{NewMail}).\\
\quad \mathit{UI}!_{f_2}\mathit{popup},\\
\mathit{Await}(f_0, f_1).\\
\quad \text{React } f_0.\\
\quad \text{Get } f_1\ (\texttt{NoMail})
\end{array}\right\}.\\
&)^*.\\
&\text{Put } f_0
\end{aligned}
\tag{1}
$$

To demonstrate the extraction of prefixes, we also moved the service's suspension into the branching type. The shared prefix of equivalent behavior in all branches has been marked in green, the retrieval of results of f_1 in yellow and differing behavior after reading the result in red.

This is the result of projecting component (1) on f_0:

$$
\begin{aligned}
&\mathit{project}^{\mathit{NotificationService}}_{f_0}((1)) =\\
&\quad \mathit{Await}(f_0, f_1).\\
&\quad \text{Get } f_1.\\
&\quad \&_{f_1}\left\{\begin{array}{l}
\texttt{NewMail} : \mathit{UI}!_{f_2}\mathit{popup},\\
\texttt{NoMail} : \mathit{Skip}
\end{array}\right\}
\end{aligned}
$$

Future-Equivalence of Method Local Types

To determine whether branches of an Offer-Type are encoding the same behavior we need a measure of equivalence of method local session types. This measure is *future-equivalence* “$\equiv$” as defined by Figure 3.18. In [28], E. Kamburjan defined session types to be future-equivalent if the histories of all systems described by one session type can be transformed into the histories of the other session type by renaming futures. In the context of this thesis, we use a slightly more restricted version of future-equivalence based on a direct, symbolic comparison of types. However, we argue it is easier to implement and it can readily be replaced by a less restricted equivalence measure if it becomes necessary in the future.

In simple terms, we regard two session types as future-equivalent if they are symbolically equal, with the exception, that if a future is newly introduced in the same position in both types for the same called method, then those future symbols are treated as being equivalent.

Example

The following types are future-equivalent:

$$p!_{fa}m.\mathit{Await}(f, fa).\,\text{Get}\, fa.\,\text{Put}\, f$$

and

$$p!_{fb}m.\mathit{Await}(f, fb).\,\text{Get}\, fb.\,\text{Put}\, f$$

And these two types are not future-equivalent (please note, that a different method on a different actor is called in the beginning):

$$p!_{fa}m_1.\mathit{Await}(f, fa).\,\text{Get}\, fa.\,\text{Put}\, f$$

and

$$q!_{fb}m_2.\mathit{Await}(f, fb).\,\text{Get}\, fb.\,\text{Put}\, f$$

For a detailed definition of future-equivalence, see the inference rules defining it in Figure 3.18.

$$\mathit{project}^t_f(L\left\langle\sigma_{\mathit{pre}},\sigma_{\mathit{post}}\right\rangle) = \begin{cases} L & \text{if } \mathit{sub}_{\mathbb{A}}(\sigma_{\mathit{pre}})(t) = (\mathit{Active}, f, \cdot) \\ \mathit{Skip} & \text{otherwise} \end{cases} \tag{3.33}$$

$$\text{where } L \in \{p!_{f'}m, \text{Put } f', \text{Get } f', \text{Get } f'\ (C)\ , \mathit{Await}(f', f''), \mathit{Skip}\}$$

$$\mathit{project}^t_f(p?_{f'}m\left\langle\sigma_{\mathit{pre}},\sigma_{\mathit{post}}\right\rangle) = \mathit{Skip} \tag{3.34}$$

$$\mathit{project}^t_f(\text{React } f'\left\langle\sigma_{\mathit{pre}},\sigma_{\mathit{post}}\right\rangle) = \mathit{Skip} \tag{3.35}$$

$$\mathit{project}^t_f(L.L') = \begin{cases} \mathit{project}^t_f(L) & \text{if } \mathit{project}^t_f(L') = \mathit{Skip} \\ \mathit{project}^t_f(L') & \text{if } \mathit{project}^t_f(L) = \mathit{Skip} \\ \mathit{project}^t_f(L).\mathit{project}^t_f(L') & \text{otherwise} \end{cases} \tag{3.36}$$

$$\mathit{project}^t_f\ \left(\left(L^{\mathbb{C}}\left\langle\sigma_{\circlearrowright},\sigma'_{\circlearrowleft}\right\rangle\right)^*\left\langle\sigma_{\mathit{pre}},\sigma_{\mathit{post}}\right\rangle\right) =$$

$$\begin{cases} \mathit{project}^t_f(L^{\mathbb{C}}\left\langle\sigma_{\circlearrowright},\sigma'_{\circlearrowleft}\right\rangle) & \text{if} \quad \nexists m.((f,t,m),\ldots) \in \mathit{sub}_{\mathbb{F}}(\sigma_{\mathit{pre}}) \qquad (3.37) \\ & \quad \wedge \exists m.((f,t,m),\ldots) \in \mathit{sub}_{\mathbb{F}}(\sigma'_{\circlearrowleft}) \\ (\mathit{project}^t_f(L^{\mathbb{C}}\left\langle\sigma_{\circlearrowright},\sigma'_{\circlearrowleft}\right\rangle))^* & \text{if } \mathit{project}^t_f(L^{\mathbb{C}}\left\langle\sigma_{\circlearrowright},\sigma'_{\circlearrowleft}\right\rangle) \neq \mathit{Skip} \qquad (3.38) \\ \mathit{Skip} & \text{otherwise} \qquad (3.39) \end{cases}$$

$$\mathit{project}^t_f\ \left(\oplus\{p\}\ (L_i{}^{\mathbb{C}})_{i\in I}\left\langle\sigma_{\mathit{pre}},\sigma_{\mathit{post}}\right\rangle\right) = \tag{3.40}$$

$$\begin{cases} \oplus\left\{(\mathit{project}^t_{f)}(L_i{}^{\mathbb{C}})_{i\in I}\right\} & \text{if } \exists i.\,\mathit{project}^t_f(L_i{}^{\mathbb{C}}) \neq \mathit{Skip} \\ \mathit{Skip} & \text{otherwise} \end{cases} \tag{3.41}$$

$$\mathit{project}^t_f\ \left(\&_{f'}\left\{(L_i{}^{\mathbb{C}})_{i\in I}\right\}\left\langle\sigma_{\mathit{pre}},\sigma_{\mathit{post}}\right\rangle\right) =$$

$$\begin{cases} \mathit{project}^t_f(L_i{}^{\mathbb{C}}) & \text{if } \mathit{introducedInBranch}(t, f, L_i{}^{\mathbb{C}}, \sigma_{\mathit{pre}}) \qquad (3.42) \\ \mathit{fixed} & \text{if } \mathit{fixed} = \qquad (3.43) \\ & \quad \mathit{identicalBranches}(t, f, (L_i{}^{\mathbb{C}})_{i\in I}) \\ \mathit{prefix}.\,\text{Get } f'.\&_{f'}\left\{(C_i : \mathit{branch}_i)_{i\in I}\right\} & \text{if} \quad (\mathit{prefix}, (C_i, \mathit{branch}_i)_{i\in I}) = \qquad (3.44) \\ & \qquad \mathit{extractBranchLabels}(t, f, (L_i{}^{\mathbb{C}})_{i\in I}) \\ & \quad \wedge |I| > 1 \end{cases}$$

Figure 3.16.: Projection function for producing method local session types from local session types for a given future $f \in \mathcal{F}$

$$\begin{aligned}
&\mathit{introducedInBranch}(t, f, L^{\mathbb{C}}\left\langle \sigma, \sigma' \right\rangle, \sigma_{\mathit{pre}}) = \\
&\qquad \forall m.((f, t, m), \ldots) \notin \mathit{sub}_{\mathbb{F}}(\sigma_{\mathit{pre}}) \\
&\qquad \wedge \exists m.((f, t, m), \ldots) \in \mathit{sub}_{\mathbb{F}}(\sigma')
\end{aligned} \tag{3.45}$$

$$\begin{aligned}
&\mathit{extractBranchLabels}(t, f, f', ({L_i}^{\mathbb{C}})_{i \in I}) = \\
&\qquad \underset{(\mathit{prefix}, (C_i, \mathit{branch}_i)_{i \in I}) \in \mathit{Splits}}{\arg\min} (|\mathit{prefix}|)
\end{aligned} \tag{3.46}$$

$$\begin{aligned}
&\text{where } \mathit{Splits} = \{ \\
&\quad (\mathit{prefix}, (C_i, \mathit{branch}'_i)_{i \in I}) \mid \\
&\qquad \exists i_0 \in I.\, (\mathit{prefix}, \cdot, \cdot) \in \mathit{splitOnGet}({L_{i_0}}^{\mathbb{C}}, t, f, f') \\
&\qquad \wedge \forall i \in I.\, \exists \mathit{prefix}_i.\, (\\
&\qquad\qquad (\mathit{prefix}_i, C_i, \mathit{branch}_i) \in \mathit{splitOnGet}({L_i}^{\mathbb{C}}, t, f, f') \\
&\qquad\quad \wedge \mathit{prefix}_i \underset{\text{fut}}{\equiv} \mathit{prefix} \\
&\qquad\quad \wedge \forall j \in I.\, i \neq j \rightarrow C_i \neq C_j \\
&\qquad) \\
&\qquad \wedge \forall i \in I.\, \mathit{branch}'_i = \begin{cases} \mathit{branch}_i & \text{if } \mathit{branch}_i \neq \emptyset \\ \mathit{Skip} & \text{otherwise} \end{cases} \\
&\}
\end{aligned} \tag{3.47}$$

$$\begin{aligned}
&\mathit{splitOnGet}(L^{\mathbb{C}}\left\langle \sigma_{\mathit{pre}}, \sigma_{\mathit{post}} \right\rangle, t, f, f') = \\
&\begin{cases}
\{(\emptyset, C, \emptyset)\} & \text{if } \; L^{\mathbb{C}} = \text{Get } f'\,(C) \\
 & \quad \wedge \mathit{sub}_{\mathbb{A}}(\sigma_{\mathit{pre}})(t) = (\mathit{Active}, f, \cdot) \\
\quad \{(\mathit{prefix}, C, \mathit{branch}.M_R) \mid & \text{if } \; L^{\mathbb{C}} = {L_L}^{\mathbb{C}}.{L_R}^{\mathbb{C}} \\
\qquad (\mathit{prefix}, C, \mathit{branch}) = \mathit{splitOnGet}({L_L}^{\mathbb{C}}, t, f, f')\} & \quad \wedge M_L = \mathit{project}^t_f({L_L}^{\mathbb{C}}) \\
\cup \{(M_L.\mathit{prefix}, C, \mathit{branch}) \mid & \quad \wedge M_R = \mathit{project}^t_f({L_R}^{\mathbb{C}}) \\
\qquad (\mathit{prefix}, C, \mathit{branch}) = \mathit{splitOnGet}({L_R}^{\mathbb{C}}, t, f, f')\} & \\
\emptyset & \text{otherwise}
\end{cases}
\end{aligned} \tag{3.48}$$

$$\mathit{identicalBranches}(t, f, ({L_i}^{\mathbb{C}})_{i \in I}) = \begin{cases}
\mathit{fixed} & \text{if} \quad I \neq \emptyset \\
 & \quad \wedge \mathit{fixed} = \mathit{project}^t_f({L_0}^{\mathbb{C}}) \\
 & \quad \wedge \forall i \in I.\, \mathit{fixed} \underset{\text{fut}}{\equiv} \mathit{project}^t_f({L_i}^{\mathbb{C}}) \\
\mathit{Skip} & \text{if } I = \emptyset
\end{cases} \tag{3.49}$$

Figure 3.17.: Helper functions and predicates used in method projection, see Figure 3.16.

$$\frac{}{\Gamma \vdash_{\text{fut}} \mathit{Skip} \equiv \mathit{Skip}}\ \textsc{Skip} \qquad \frac{}{\Gamma \vdash_{\text{fut}} \emptyset \equiv \emptyset}\ \textsc{EmptyTypes}$$

$$\frac{\Gamma \cup \{(f, f')\} \vdash_{\text{fut}} M \equiv M'}{\Gamma \vdash_{\text{fut}} p!_f m.M \equiv p!_{f'} m.M'} \qquad \frac{\begin{array}{c} f_1 = f_1' \vee f_1\ \Gamma\ f_1' \\ f_2 = f_2' \vee f_2\ \Gamma\ f_2' \quad \Gamma \vdash_{\text{fut}} M \equiv M' \end{array}}{\Gamma \vdash_{\text{fut}} \mathit{Await}(f_1, f_2).M \equiv \mathit{Await}(f_1', f_2').M'}$$

$$\frac{f = f' \vee f\ \Gamma\ f' \quad \Gamma \vdash_{\text{fut}} M \equiv M'}{\Gamma \vdash_{\text{fut}} \text{Get}\ f\ [(C)]\ .M \equiv \text{Get}\ f'\ [(C)]\ .M'} \qquad \frac{f = f' \vee f\ \Gamma\ f' \quad \Gamma \vdash_{\text{fut}} M \equiv M'}{\Gamma \vdash_{\text{fut}} \text{Put}\ f\ [(C)]\ .M \equiv \text{Put}\ f'\ [(C)]\ .M'}$$

$$\frac{\Gamma \vdash_{\text{fut}} M_{\circlearrowleft} \equiv M'_{\circlearrowleft} \quad \Gamma \vdash_{\text{fut}} M \equiv M'}{\Gamma \vdash_{\text{fut}} (M_{\circlearrowleft})^*\ .M \equiv (M'_{\circlearrowleft})^*\ .M'}\ \textsc{Repeat}$$

$$\frac{\begin{array}{c} \forall i \in I.\, \exists j \in J.\, \Gamma \vdash_{\text{fut}} M_i.M \equiv M_j'.M' \\ \forall j \in J.\, \exists i \in I.\, \Gamma \vdash_{\text{fut}} M_i.M \equiv M_j'.M' \end{array}}{\Gamma \vdash_{\text{fut}} \oplus \left\{(M_i)_{i \in I}\right\} .M \equiv \oplus \left\{(M_j')_{j \in J}\right\} .M'}\ \textsc{Choose}$$

$$\frac{f = f' \vee f\ \Gamma\ f' \quad \begin{array}{c} \forall i \in I.\, \exists j \in J.\, C_i = C_j' \\ \forall j \in J.\, \exists i \in I.\, C_i = C_j' \end{array} \quad \forall i \in I.\, \forall j \in J.\, C_i = C_j' \implies \Gamma \vdash_{\text{fut}} M_i.M \equiv M_j'.M'}{\Gamma \vdash_{\text{fut}} \&_f \left\{(C_i : M_i)_{i \in I}\right\} .M \equiv \&_{f'} \left\{(C_j' : M_j')_{j \in J}\right\} .M'}\ \textsc{Offer}$$

Figure 3.18.: Definition of future-equivalence for Method Local Types.

3.4. Static Verification

A global session type specifies a possibly branching or repeating sequence of communication and scheduling actions for every method of a set of methods, see also Section 3.3.2. The objective of static verification in this thesis is to automatically verify that every method of an ABS model implements its specified actions in the right order.

For this purpose, we check the ASTs of methods using a type system, see Section 3.4.3. The subset of the ABS language we are operating on is presented in Section 3.4.1. Our mapping from actors in a session type to their representation in an AST is explained in Section 3.4.2. Finally, we discuss the soundness of our type system relative to the original one developed by Kamburjan et al. [30] in Section 3.4.5.

Exceptions and Assertions

To simplify static verification, we assume that no exceptions are ever thrown by a method and no assertions are violated[1]. If an exception was thrown, this could prevent the progress of a session. However, we at least ensure that an exception can not alter the control flow of a method since our type system does not allow **`try-catch`** statements. We incorporated this assumption into our soundness theorems (see Sections 3.4.5 and 3.5.8) and liveness guarantees (see Section 3.6).

3.4.1. Kernel Language

The SDS-Tool does work with ASTs produced by the abstools compiler for the currently implemented version of the ABS language (as of 2019-10-08). Therefore, it supports more language features than Core ABS [27] (e. g. **`case`**-statements) but also not the full ABS language implementation.

Moreover, it only inspects those parts of an ABS model referenced in the session type specification used for verification. For example, the **`suspend`**-statement causes a computation to unconditionally suspend its execution and such an action can not be specified in a session type. Thus, if **`suspend`** is used in a method m which is specified to be called in the session type, verification will result in an error. Yet, if m is never mentioned in the type and therefore has no part in the specified session, m is never inspected and no error

[1] If an assertion is violated, this results in an exception.

is raised. Of course, parts of the model not mentioned in the type might interfere with a session if not checked, we call this a half-open system, see Section 3.4.4 and Section 3.6.

In Figure 3.19 we define a subset of the ABS language which includes all features relevant to the verification process. If an ABS feature not covered by this language definition is encountered during verification, it raises an error. Notably, **suspend**-statements and synchronous calls e_p`.m(...)` are not included. We also omitted the definition of some language features like pure expressions e_p for brevity. They can be looked up in Deliverable 1.2 of the HATS project [11].

$$
\begin{array}{rcl}
\boxed{P} & ::= & \boxed{\overline{Dd}}\ \boxed{\overline{F}}\ \boxed{\overline{In}}\ \boxed{\overline{Cl}}\ [\boxed{\overline{B}}] \\
\boxed{In} & ::= & \texttt{interface}\ I\ [\texttt{extends}\ \ I^{+}]\ \ \{\ \boxed{\overline{M_s}}\ \} \\
\boxed{Cl} & ::= & \texttt{class}\ C\ [\overline{\boxed{T}\,fd}]\ [\texttt{implements}\ \ I^{+}]\ \ \{\ \overline{\boxed{T}\ fd\ [\texttt{=}\ \ \boxed{e_p}]}\ [\boxed{\mathrm{B}}]\ \boxed{\overline{M}}\ \} \\
\boxed{M} & ::= & \boxed{M_s}\ \boxed{B} \\
\boxed{M_s} & ::= & \boxed{T}\ m\ (\ \overline{\boxed{T}\ x}\ ) \\
\boxed{B} & ::= & \{\ \overline{\boxed{s}}\} \\
\boxed{T} & ::= & \boxed{I}\ \mid\ \boxed{D}\ \mid\ \texttt{Fut<}\boxed{T}\texttt{>} \\
\boxed{v} & ::= & x\ \mid\ \texttt{this}.fd \\
\boxed{e} & ::= & \boxed{e_p}\ \mid\ \boxed{e_e} \\
\boxed{e_e} & ::= & \texttt{new}\ C(\boxed{\overline{e_p}})\ \mid\ \boxed{e_p}\texttt{!}m(\overline{\boxed{e_p}})\ \mid\ \boxed{e_p}\texttt{.get} \\
\boxed{s} & ::= & \boxed{v}\ \texttt{=}\ \boxed{e}\texttt{;}\ \mid\ \texttt{await}\ \boxed{g}\texttt{;}\ \mid\ \texttt{skip;}\ \mid\ \boxed{e}\texttt{;}\ \mid\ \texttt{case}\ \boxed{e_p}\{\ \boxed{\overline{b}}\}\ \mid \\
 & & \texttt{if}\ \ (\boxed{e_p})\ \boxed{B}\ [\texttt{else}\ \boxed{B}]\ \mid\ \texttt{while}\ \ (\boxed{e_p})\ \boxed{B}\ \mid\ \boxed{B}\ \mid\ \boxed{T}\ \boxed{v}\ \texttt{=}\ \boxed{e}\texttt{;}\ \mid \\
 & & \texttt{return}\ \boxed{e}\texttt{;}\ \mid\ \boxed{T}\ \boxed{v}\texttt{;}\ \mid\ \texttt{assert}\ \boxed{e_p}\texttt{;}\ \mid\ \texttt{duration(}\boxed{e_p}\texttt{,}\ \boxed{e_p}\texttt{);} \\
\boxed{g} & ::= & \boxed{v}\texttt{?}\ \mid\ \boxed{g}\ \texttt{\&}\ \boxed{g}\ \mid\ \boxed{e_p} \\
\boxed{b} & ::= & \boxed{p}\ \texttt{=>}\ \boxed{s} \\
\boxed{p} & ::= & \_\ \mid\ Co(\boxed{\overline{p}})\ \mid\ \boxed{p_{+}}
\end{array}
$$

Figure 3.19.: ABS sub-language relevant to the verification process. I denotes an interface identifier, *fd* a field identifier, m a method identifier, x a local variable or parameter identifier and *Co* constructor names.

3.4.2. Actor Representation in ABS

Every instance of a class which is the only member of its COG is an active object as defined in Section 2.1.1. For verification, we must have a mapping from such instances o in a model to an actor symbol p of a session type s. We say "o performs *role* p in s". The following complications must be considered:

- Given an object reference in a method context, it is non-trivial to decide whether it points to an object mapped to a specific actor.
- Dynamic enforcement, see Section 3.5 requires AST modifications on the class level. Thus, it affects all instances of a class.
- In ABS, objects can only be referenced by their interfaces and interfaces can be shared by multiple classes.

Thus we decided on the following restrictions on ABS models for them to be verifiable. They avoid elaborate points-to analyses and permit class-wide AST modifications:

1. Roles of a session type must be implemented as classes.
2. A class performing a role must have exactly one instance in the whole model which is created in the main block.
3. Classes implementing roles may not share interfaces.
4. A class implementing a role must have the same name as the role.

This way, a mapping from interface identifiers to actor symbols can be created for static verification. The computation of this mapping has been formalized as a function *interfaceMapping* in Figure 3.20.

Please note, that requirement 2 is not thoroughly verified by the type system introduced in the next section. Instead, Section 3.4.4 presents a separate strategy to enforce it.

$$
\begin{aligned}
&\mathit{interfaceMapping}\colon 2^{\mathit{Actors}} \times \mathit{ClassDecls}^{*} \times \mathit{InterfaceDecls}^{*} \rightarrow (\mathit{Interfaces} \rightharpoonup \mathit{Actors})\\
&\mathit{interfaceMapping}(A, \overline{Cl}, \overline{In}) = \{(I, C) \mid\\
&\qquad (C, \mathit{Is}) \in \mathit{classInterfaces}(\overline{Cl}, \overline{In})\\
&\quad \wedge C \in A\\
&\quad \wedge I \in \mathit{Is}\\
&\quad \wedge \nexists (C', \mathit{Is}') \in \mathit{classInterfaces}(\overline{Cl}, \overline{In}).\ (\\
&\qquad\quad C' \in A\\
&\qquad \wedge \mathit{Is} \cap \overline{\mathit{In}} \neq \emptyset\\
&\quad)\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{classInterfaces}(\overline{Cl}, \overline{In}) = \{(C, \mathit{Is}) \mid\\
&\qquad \boxed{\texttt{class C} \ \ldots\ \texttt{implements}\ I_1, I_2, \ldots, I_n\ \ \{\ldots\}} \in \overline{Cl}\\
&\quad \wedge \mathit{Is} = \{I_1, I_2, \ldots, I_n\} \cup \bigcup_{I' \in \{I_1, I_2, \ldots I_n\}} \mathit{superInterfaces}(I', \overline{\mathit{In}})\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{superInterfaces}(I, \overline{\mathit{In}}) = I_{\mathit{direct}} \cup \bigcup_{I' \in I_{\mathit{direct}}} \mathit{superInterfaces}(I', \overline{\mathit{In}})\\
&\quad \text{where } I_{\mathit{direct}} = \{I_1, I_2, \ldots I_n \mid\\
&\qquad\quad \boxed{\texttt{interface}\ I\ \texttt{extends}\ I_1, I_2, \ldots I_n\ \ \{\ldots\}} \in \overline{\mathit{In}}\\
&\quad \}
\end{aligned}
$$

Figure 3.20.: Formalization of the mapping from ABS interface identifiers to session type roles.

3.4.3. Type System

Static verification is performed using a type system expressed by a set of inference rules, see Figures 3.21 to 3.23. It bears some similarities to the type systems used for verification by Kamburjan et al. in [30] and [29]. However, we leave the task of data type checking to the *abstools* compiler and focus solely on the verification of a session type. Our type system also aims to be more practical, that is, we try to give programmers more freedom in designing their models if possible. For example, all statements and control structures which do not influence or violate a session are ignored during verification, see rule CommInert etc.

The rules in Figure 3.21 apply to the structure of an ABS model: Classes, methods and the main block. Figures 3.22 and 3.23 operate on the contents of methods i. e. sequences of statements, whereas Figure 3.22 contains all the axiomatic rules. We write

$$\overline{e} : s$$

to say the AST elements $\overline{e}$ comply with session type s. The judgments within the rules use the following environments:

Δ: *I* $\rightharpoonup$ *Actors* Environment Δ maps interface identifiers to actor names. It is first introduced in rule Model using the function *interfaceMapping*, see above section.

$F \subseteq \mathcal{F}$ F contains all future symbols used in the session type specification for which we verify. It is derived from the information gathered by analysis $\mathbb{F}$.

Γ: $\mathcal{F} \rightharpoonup$ *VarIdent* Γ maps futures to identifiers of local variables which have been used to store the futures' results.

Structural Rules

Verification always starts with the rule Model being applied to the whole ABS model. It checks that no interfaces of classes with a role intersect by making sure that every actor is accessible through Δ. Further verification is delegated to the rules Classes and Main.

Rule Main checks whether there is an asynchronous call $e_p\,!\,m(\overline{e'_p})$ in the main block implementing the initial action $0 \xrightarrow{f} q\colon m$ of the session type. This also requires that the callee expression of the call is an object whose interface is mapped to q in environment Δ. We write *interface*(e_p) to retrieve the identifier of the interface that types e_p. Main also checks that there are no other calls to actors of the session type ($\emptyset = \ldots$) and that all classes with roles are instantiated as their own COG. The last part is symbolized by the function *checkForNew*.

Rule Classes verifies that for every actor p in the session type there is a class *Cl′* in the model taking up its role (same name). Further verification is delegated to the Class rule. It operates on the projection of the session type on p, as *Cl′* only needs to implement the behavior of p. Since the rule Method needs to know which actor is described by the local type, p is passed on as an environment.

The Class rule ensures that all methods of a class conform to the session type. Please note, that the rule only accepts classes without an init-Block or run-Method. An init-Block, if present, is executed synchronously after a class has been instantiated and a run-Method asynchronously after the init-Block finishes. Both kinds of behavior can not be encoded in our session type language and are thus forbidden for all classes taking up a role[2].

For all futures f targeting a method m of an actor p, rule Method requires m's body to exert the behavior described in the session type for f. This behavior description is extracted by method projection. If there is no future introduced by a call to m, the method is ignored by the verification process.

Statement Rules

The remaining rules verify the behavior implemented by the bodies of methods. Their underlying principle is that for all statements influencing communication and cooperative scheduling there must be a corresponding action described in the type. Also, the order of actions must be adhered. All statements which do not exercise such influence are ignored which is expressed by rule CommInert. The predicate $\mathit{commInert}_{\Gamma;F;\Delta}$ decides whether a statement or expression falls under this category. It has been formalized in Figure 3.25. For a sequence $\overline{s}$ of statements we write $\mathit{commInert}_{\Gamma;F;\Delta}(\overline{s})$ to convey that every statement of the sequence fulfills the predicate.

Rules ReturnNoMsg, MethodEnd and ReturnMsg are all axiomatic rules which apply to the end of a method. Therefore, they require the session type to be a resolving Put f type and be the last action specified. A method can either end if there are no more statements, which is handled by MethodEnd, or it can end with a **return**-statement, which is handled by the other two rules. We require the expression of a **return** e; statement to not have any effects on communication or scheduling, like a call to another actor: $\mathit{commInert}_{\Gamma;F;\Delta}(e)$. If a session type specifies a constructor, we also check whether the type

[2] ABS classes also support so-called recovery-Blocks which can handle uncaught exceptions. We also forbid those.

constructed by it is the same as the type of the return expression e: *typeOf*$(e) =$ *typeOf*(C). This is not precise, since another constructor of the type could have been used to construct the expression. However, for a more precise check another analysis of control flow etc. would be required to determine how e has been computed. Because of the time constraints applying to this thesis we leave development of this check for future work. Meanwhile, even if a method implementation does not use the constructor specified in the type for its return value, the remaining model execution will still continue as specified. Notably, control structures implementing Offer-Types must still check values against constructors as specified, see rule Offer.

Rule TypeEnd is the last axiomatic rule. It applies if there remain no actions in the specification (which is represented by "$\emptyset$") and all remaining statements have no effect on communication or scheduling. The rule is necessary to handle nested types which do not end with a resolving action, for example whenever rule While is applied:

$$\dfrac{\dfrac{\dots \quad \dfrac{\dots \quad \dfrac{}{\Gamma; F; \Delta \vdash \emptyset : \emptyset}\ \text{TypeEnd}}{\Gamma; F; \Delta \vdash \texttt{this.f = o!m();} : p!_f m}\ \text{Call}}{}}{\Gamma; F; \Delta \vdash \texttt{while (i < n) \{ this.f = o!m(); \}} : (p!_f m)^* \,.\, \mathrm{Put}\, f}\ \text{While}$$

It can not be used to verify the end of a method, since all method local session types end with a resolving action which can only be handled by the other three axiomatic rules.

The remaining non-axiomatic rules allow the verification to process a method body statement by statement. Rule Call verifies interactions with other objects. To keep the verification process simple, we decided on the following restrictions:

1. futures produced by an interaction of a session must be stored in fields so that the future is accessible to other methods of the object.
2. futures must be stored in fields of the same name as the future symbol in the type. Otherwise, we would have to track across methods where a future has been stored.
3. If there is a future f in the type, a field **this**.f can only be written by the statement which issues the call producing future f.

Rules Call and FieldAssign ensure these restrictions. Please note that unchecked methods not participating in a session which could potentially modify fields are no issue, since the schedulers created by our dynamic enforcement (Section 3.5) strategies, prevent them from being executed. See also the safety guarantee in Section 3.6. Similar to rule Main, the environment Δ is used in rule Call to verify that the callee is the one mentioned in the specification.

Rule FieldAssign complements Call; that is, it requires that no fields with future names are written if its not an interaction of a session. Also, an expression assigned to a field then may have no side-effects on communication or scheduling.

For storing the results of fetching actions Get $f\ [(C)]$ similar concerns as with futures apply. We decided against demanding to reserve fields for storing **get**-expressions and to use local variables instead. While there may be only one call statement per future introduction, fetching actions may be specified an unlimited number of times requiring reservation of multiple fields. Therefore, local variables are more practical than fields in this case, especially since futures are already accessible for **get**-expressions across methods. These are the requirements on fetching actions as checked by rules Get and VarAssign:

1. **get**-expressions must be stored in local variables. (See rule Get.)
2. This can be during a variable declaration or by assignment to an existing variable. We remember the location per future in environment Γ. (See rule Get.)
3. after such an assignment or declaration, the variable may never be assigned again. (See rule Get and rule VarAssign.)
4. in half-open systems (see Section 3.6) **get**-expressions can halt the execution of a method depending on an actor which is not part of a session. Therefore, **get**-expressions are permitted only if they are part of the specification. Hence, the predicate $\mathit{commInert}_{\Gamma;F;\Delta}$ is never fulfilled for **get**-expressions.

If the specification requires a method to release control until another future is ready, rule Await checks that its body contains an **await**-statement waiting on a field with the name of the awaited future.

Rule Block unpacks blocks so that they can be analyzed by the other rules.

Since the *Skip* type specifies no action, rule Skip allows the verification to process it by continuing with the remaining type.

Rule While checks, that repetition types are implemented by a **while**-loop. It verifies the loop body against the nested type and the remaining statements against the remainder of the type. Since there may be loops whose statement block has no influence on communication or scheduling within a session, those must be handled by rule CommInert. We prevent the use of rule While on them by checking the body against the predicate $\mathit{commInert}_{\Gamma;F;\Delta}$. Please note, that this makes it impossible to verify session types of the form $(\mathit{Skip})^*$, but those are never produced by projection.

For the verification of Offer-Types, we decided to only allow implementations which make use of a non-nested **case**-statement. This is because rule Offer can then simply match the labels of its branches against constructor patterns of the statement. To comply with a Offer Type $\&_f \{\dots\}$, a **case**-statement must use a local variable as input expression which stores the result of f since f decides which branch to take. This information is gained from the environment Γ: $v \in \Gamma(f)$. The Offer rule also requires that there are as many unique constructor patterns as there are labels (same index set I). For each matching pair of constructor case pattern and label ($C'_i = C_j$), the case's body s_i and the statements $\overline{s}$ after the **case**-construct must adhere to the specification M_i belonging to the label and the remainder of the type M: "$s_i.\overline{s} : M_j.M$". We do not check that the constructor case patterns are exhaustive for the data type of the checked expression. However, if they are not, an exception is thrown at runtime when encountering a value which fits none of the patterns. We regard this as an instance of runtime-verification, see also Section 3.7. Analogously to the While rule, Offer is only applicable to **case**-statements with at least one branch affecting communication or scheduling within the session: $\neg\mathit{commInert}_{\Gamma;F;\Delta}(s_i)$.

Rule Choice is applicable, if there is one type branch M_i specifying the behavior of the given statements $s_1\overline{s_2}$. Also, all leading control flow structures containing statements influencing the session must first be resolved by the IfNoOffer, IfElseNoOffer and CaseNoOffer rules. These ensure that such control flows are checked against the specification, eventually leading up to an application of the Choice rule:

$$\dfrac{\dots \quad \dfrac{\dots \quad \dfrac{\dots}{\Gamma;F;\Delta \vdash \texttt{this.f = o!m();} : p!_f m.M}\,\textsc{Call}}{\Gamma;F;\Delta \vdash \texttt{this.f = o!m();} : \oplus\{p!_f m, \mathit{Skip}\}.M'}\,\textsc{Choice} \qquad \dfrac{\dots \quad \dfrac{\dots}{\Gamma;F;\Delta \vdash \texttt{skip;} : \mathit{Skip}.M'}\,\textsc{CommInert}}{\Gamma;F;\Delta \vdash \texttt{skip;} : \oplus\{p!_f m, \mathit{Skip}\}.M'}\,\textsc{Choice}}{\Gamma;F;\Delta \vdash \texttt{if (True) \{this.f = o!m();\} else \{skip;\}} : \oplus\{p!_f m, \mathit{Skip}\}.M'}\,\textsc{IfElseNoOffer}$$

Those rules can however not be used if the first component of the session type is an Offer-Type, because those may only be implemented by a specific kind of **case**-statement, as mentioned.

$$\dfrac{\begin{array}{l}\Delta = \mathit{interfaceMapping}(\mathit{sub}_{\mathbb{P}}(\sigma_{post}), \overline{\mathit{Cl}}, \overline{\mathit{In}}) \\ \quad \forall p \in \mathit{sub}_{\mathbb{P}}(\sigma_{post}).\, \exists I.\, \Delta(I) = p \qquad \Delta \vdash_{\text{Classes}} \overline{\mathit{Cl}} : G^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle \quad \Delta \vdash_{\text{Main}} B : G^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle\end{array}}{\vdash_{\text{Model}} \overline{\mathit{Dd}}\ \overline{F}\ \overline{\mathit{In}}\ \overline{\mathit{Cl}}\ B : G^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle}\ \textsc{Model}$$

$$\dfrac{\forall p \in \mathit{sub}_{\mathbb{P}}(\sigma_{post}).\, \exists \mathit{Cl}' \in \overline{\mathit{Cl}}.\, (p = \mathit{name}(\mathit{Cl}') \wedge p; \Delta \vdash_{\text{Class}} \mathit{Cl}' : \mathit{project}^{p}(G^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle))}{\Delta \vdash_{\text{Classes}} \overline{\mathit{Cl}} : G^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle}\ \textsc{Classes}$$

$$\dfrac{\forall m \in \overline{M}.\, p; \Delta \vdash_{\text{Method}} m : L^{\mathbb{C}} \qquad T\ \texttt{run(}\ldots\texttt{)}\ \texttt{\{}\ldots\texttt{\}} \notin \overline{M}}{p; \Delta \vdash_{\text{Class}} \texttt{class}\ C\ [(\overline{T\ f})]\ [\texttt{implements}\ \mathit{If}^{+}]\ \{[\overline{T\ f[= e_p]}]\ \overline{M}\} : L^{\mathbb{C}}}\ \textsc{Class}$$

$$\dfrac{\begin{array}{c}F = \{f \mid (f', \cdot, \cdot) \in \mathit{sub}_{\mathbb{F}}(\sigma_{post})\} \\ \forall (f, p, m) \in \mathit{sub}_{\mathbb{F}}(\sigma_{post}).\, \emptyset; F; \Delta \vdash \overline{s} : \mathit{project}^{p}_{f}(L^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle)\end{array}}{p; \Delta \vdash_{\text{Method}} T\ m(\overline{T\ x})\ \{\overline{s}\} : L^{\mathbb{C}}\,\langle \sigma_{pre}, \sigma_{post}\rangle}\ \textsc{Method}$$

$$\dfrac{\begin{array}{ll}\Delta(\mathit{interface}(e_p)) = q & \qquad \mathit{checkForNew}(\mathit{sub}_{\mathbb{P}}(\sigma_{post}), \overline{s_L}, \overline{s_R}) \\ s = [[T]v\texttt{=}]e_p\,!\,m(\overline{e'_p}) & \emptyset = \mathit{sub}_{\mathbb{P}}(\sigma_{post}) \cap \left\{\Delta(\mathit{interface}(e''_p)) \mid e''_p\,!\,m''(\ldots) \in \mathit{Calls}(\overline{s_L}) \cup \mathit{Calls}(\overline{s_R})\right\}\end{array}}{\Delta \vdash_{\text{Main}} \overline{s_L}\ s\ \overline{s_R} : 0 \xrightarrow{f} q\colon m\,\langle \cdot, \cdot\rangle\,.G\,\langle \cdot, \sigma_{post}\rangle}\ \textsc{Main}$$

Figure 3.21.: Rules of the type system for static verification (Part 1).

$$\dfrac{\mathit{commInert}_{\Gamma;F;\Delta}(e)}{\Gamma; F; \Delta \vdash \texttt{return}\ e\texttt{;} : \mathrm{Put}\ f}\ \textsc{ReturnNoMsg} \qquad \dfrac{}{\Gamma; F; \Delta \vdash \emptyset : \mathrm{Put}\ f}\ \textsc{MethodEnd}$$

$$\dfrac{\mathit{commInert}_{\Gamma;F;\Delta}(e) \quad \mathit{typeOf}(e) = \mathit{typeOf}(C)}{\Gamma; F; \Delta \vdash \texttt{return}\ e\texttt{;} : \mathrm{Put}\ f[C]}\ \textsc{ReturnMsg} \qquad \dfrac{\mathit{commInert}_{\Gamma;F;\Delta}(\overline{s})}{\Gamma; F; \Delta \vdash \overline{s} : \emptyset}\ \textsc{TypeEnd}$$

Figure 3.22.: Axiomatic rules of the type system for static verification of method bodies (Part 2).

$$\frac{\Delta(\mathit{interface}(e_p)) = q \quad \Gamma; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash \mathtt{this}.f\ \mathtt{=}\ e_p\,\mathtt{!}\,m\mathtt{(}\overline{e_p}\mathtt{)}\mathtt{;}\ \overline{s} : q!_f m.M}\ \textsc{Call}$$

$$\frac{\mathit{fd} \notin F \quad \mathit{commInert}_{\Gamma;F;\Delta}(e) \quad \Gamma; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash \mathtt{this}.\mathit{fd}\ \mathtt{=}\ e\mathtt{;}\ \overline{s} : M}\ \textsc{FieldAssign}$$

$$\frac{\forall f'.\, v \notin \Gamma(f') \quad \Gamma[f \coloneqq \Gamma(f) \cup \{v\}]; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash [T]\ \ v\ \mathtt{=}\ \mathtt{this}.f.\mathtt{get}\mathtt{;}\ \overline{s} : \mathrm{Get}\, f\, [(C)]\, .M}\ \textsc{Get}$$

$$\frac{\forall f'.\, v \notin \Gamma(f') \quad \mathit{commInert}_{\Gamma;F;\Delta}(e) \quad \Gamma; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash v\ \mathtt{=}\ e\mathtt{;}\ \overline{s} : M}\ \textsc{VarAssign}$$

$$\frac{\Gamma; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash \mathtt{await}\ \mathtt{this}.f'\mathtt{?;}\ \overline{s} : \mathit{Await}(f, f').M}\ \textsc{Await}$$

$$\frac{\Gamma; F; \Delta \vdash \overline{s} : M}{\Gamma; F; \Delta \vdash \overline{s} : \mathit{Skip}.M}\ \textsc{Skip} \qquad\qquad \frac{\Gamma; F; \Delta \vdash \overline{s}\ \overline{s'} : M}{\Gamma; F; \Delta \vdash \{\overline{s}\}\ \ \overline{s'} : M}\ \textsc{Block}$$

$$\frac{\mathit{commInert}_{\Gamma;F;\Delta}(s_1) \quad \Gamma; F; \Delta \vdash \overline{s_2} : M}{\Gamma; F; \Delta \vdash s_1 \overline{s_2} : M}\ \textsc{CommInert}$$

Figure 3.23.: Rules of the type system for static verification of method bodies (Part 3).

$$\dfrac{\neg \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s_1}) \quad \Gamma;F;\Delta \vdash \overline{s_1} : M \quad \Gamma;F;\Delta \vdash \overline{s_2} : M}{\Gamma;F;\Delta \vdash \texttt{while (}e_p\texttt{) \{}\overline{s_1}\texttt{\}}\ \overline{s_2} : (M_1)^{*}.M_2}\ \textsc{While}$$

$$\dfrac{\begin{array}{c} \forall i \in I.\, \forall j \in I.\, (i \neq j \implies C'_i \neq C'_j) \\ v \in \Gamma(f) \quad \exists i \in I.\, \neg \mathit{commInert}_{\Gamma;F;\Delta}(s_i) \quad \forall i \in I.\, \exists j \in I.\, (C'_i = C_j \wedge \Gamma;F;\Delta \vdash s_i\ \overline{s} : M_j.M) \end{array}}{\Gamma;F;\Delta \vdash \texttt{case}\ v\ \texttt{\{(}C'{}_i(\overline{p_i})\ \texttt{=>}\ s_i)_{i\in I}\texttt{\}}\ \overline{s} : \&_f \left\{(C_i : M_i)_{i\in I}\right\}.M}\ \textsc{Offer}$$

$$\dfrac{\begin{array}{c} \exists i \in I.\, \Gamma;F;\Delta \vdash s_1\overline{s_2} : M_i.M \\ \neg \mathit{commInert}_{\Gamma;F;\Delta}(s_1) \quad s_1 \text{ is neither an } \texttt{if} - \text{ nor } \texttt{case}\text{-statement} \end{array}}{\Gamma;F;\Delta \vdash s_1\overline{s_2} : \oplus\left\{(M_i)_{i\in I}\right\}.M}\ \textsc{Choice}$$

$$\dfrac{\begin{array}{c} \Gamma;F;\Delta \vdash \overline{s_1}\ \overline{s_2} : M.M' \quad \Gamma;F;\Delta \vdash \overline{s_2} : M.M' \\ \neg \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s_1}) \qquad M \text{ begins not with an Offer-Type} \end{array}}{\Gamma;F;\Delta \vdash \texttt{if (}e_p\texttt{) \{}\overline{s_1}\texttt{\}}\ \overline{s_2} : M.M'}\ \textsc{IfNoOffer}$$

$$\dfrac{\begin{array}{c} \Gamma;F;\Delta \vdash \overline{s_1}\ \overline{s_3} : M.M' \quad \Gamma;F;\Delta \vdash \overline{s_2}\ \overline{s_3} : M.M' \\ \neg(\mathit{commInert}_{\Gamma;F;\Delta}(\overline{s_1}) \vee \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s_2})) \qquad M \text{ begins not with an Offer-Type} \end{array}}{\Gamma;F;\Delta \vdash \texttt{if (}e_p\texttt{) \{}\overline{s_1}\texttt{\} else \{}\overline{s_2}\texttt{\}}\ \overline{s_3} : M.M'}\ \textsc{IfElseNoOffer}$$

$$\dfrac{\begin{array}{c} \exists i \in I.\, \mathit{commInert}_{\Gamma;F;\Delta}(s_i) \quad \forall i \in I.\, \Gamma;F;\Delta \vdash s_i\ \overline{s} : M.M' \\ M \text{ begins not with an Offer-Type} \end{array}}{\Gamma;F;\Delta \vdash \texttt{case}\ e_p\ \texttt{\{(}p_i\ \texttt{=>}\ s_i)_{i\in I}\texttt{\}}\ \overline{s} : M.M'}\ \textsc{CaseNoOffer}$$

Figure 3.24.: Rules of the type system for static verification of method bodies (Part 4).

$$
\begin{aligned}
&\mathit{commInert}_{\Gamma;F;\Delta} : (\mathit{Statement} \cup \mathit{Expression}) \rightarrow \mathbb{B}\\
&\mathit{commInert}_{\Gamma;F;\Delta}(x) = \\
&\qquad x = \texttt{skip;}\\
&\quad \vee\, x = \texttt{assert ...;}\\
&\quad \vee\, x = \texttt{duration(...);}\\
&\quad \vee\, x \text{ is an ABS pure expression}\\
&\quad \vee\, x = e\texttt{;} \wedge e \text{ is an expression } \wedge \mathit{commInert}_{\Gamma;F;\Delta}(e)\\
&\quad \vee\, x = \texttt{\{}\overline{s}\texttt{\}} \wedge \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s})\\
&\quad \vee\, x = \texttt{if (...) \{}\overline{s}\texttt{\}} \wedge \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s})\\
&\quad \vee\, x = \texttt{if (...) \{}\overline{s_1}\texttt{\} else \{}\overline{s_2}\texttt{\}} \wedge \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s_1}) \wedge \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s_2})\\
&\quad \vee\, x = \texttt{case ... \{}(p_i \texttt{ => } s_i)_{i \in I}\texttt{\}} \wedge \forall i \in I.\, \mathit{commInert}_{\Gamma;F;\Delta}(s_i)\\
&\quad \vee\, x = \texttt{while (...) \{}\overline{s}\texttt{\}} \wedge \mathit{commInert}_{\Gamma;F;\Delta}(\overline{s})\\
&\quad \vee\, x = T\ \texttt{x = } e\texttt{;} \wedge \mathit{commInert}_{\Gamma;F;\Delta}(e)\\
&\quad \vee\, x = x\ \texttt{= } e\texttt{;} \wedge \mathit{commInert}_{\Gamma;F;\Delta}(e) \wedge \nexists f.\, x \in \Gamma(f)\\
&\quad \vee\, x = \texttt{this.}\mathit{fd}\texttt{ = } e\texttt{;} \wedge \mathit{commInert}_{\Gamma;F;\Delta}(e) \wedge \mathit{fd} \notin F\\
&\quad \vee\, x = e_p\texttt{!m(...)} \wedge \nexists p.\, p = \Delta(\mathit{interface}(e_p))\\
&\quad \vee\, x = \texttt{new } C\texttt{(...)} \wedge \nexists p.\, p = \Delta(\mathit{interface}(\texttt{new } C\texttt{(...)}))\\
&\quad \vee\, x = \emptyset
\end{aligned}
$$

Figure 3.25.: Predicate to check, whether a statement or expression does not influence communication or cooperative scheduling within a session.

3.4.4. Additional Verification Steps

As mentioned, the introduced type system does only inspect those parts of an ABS model directly referenced in a session type. Thus, the following issues arise:

1. Classes implementing actors may be instantiated outside the main block in some class which is never inspected.
2. Non-inspected parts of a model might communicate with members of a session and disrupt it, though we give a basic safety guarantee. See Section 3.6.

Issue 1 is checked statically. The implementation of our tool searches the model AST for **new**-expressions. If any instantiations of classes having a role in a session type are spotted outside of the main block, verification fails. Adherence to the requirement of a single instantiation within the main block is handled by the type system.

3.4.5. Method-Local Soundness of our Type System

Kamburjan et al. present a type system in [30] and [28] which verifies that methods of an ABS model comply locally with a global session type specification. By extension it also verifies that the entire model globally complies with the type if (re-)activations of methods are scheduled in the specified order.

We informally argue that our static verification process exerts the same qualities for the following reasons:

Similar specification language. Our session types specify the same actions as theirs. There are only superficial differences. For example, we eliminated the **end** type. Instead, the syntactical end of the type implies the termination of all actions. Their global types also contain no *Skip* type in constrast to ours. This is only relevant for branching types where *Skip* can be used to express that a branch contains no actions:

$$G_a.p \begin{Bmatrix} G_b, \\ \mathit{Skip} \end{Bmatrix}$$

However, the same can be expressed in their Session Type language by extending all branchings until the end of the type and placing their **end** type in one branch:

$$G_a.p \begin{Bmatrix} G_b.\mathbf{end}, \\ \mathbf{end} \end{Bmatrix}$$

Stricter validation process The validation performed by our CST Validation process permits fewer specifications than are accepted by their projection. Therefore, the accepted input language of our static verification process is a subset of theirs. For example, we have a notion of scope for future symbols which requires future symbols to be resolved within the branch or repetition they have been created in, whereas there is no such restriction on their types.

Moreover, all important checks, e. g. self-containedness of repeated types, are performed.

Similar projection Though our projection process does strongly differ in some instances from theirs, mostly due to the use of our CST Validation, we argue that the resulting object and method types also extract all actions relevant to the target object and preserve their order.

Type system based on same principles Our type system like theirs does compare every method participating in the specification against its method local type. All simple actions derived from $\mathcal{G}_{atomic}$ must be matched by a corresponding statement in the right order. More complex types have to be implemented with corresponding control structures.

Our system does however prominently differ in two points, but our objective of static verification is not compromised due to the stated reasons:

1. We perform no data type checking, but it is delegated to the ABS compiler.
2. More kinds of statements are allowed by the type system for practical reasons (CommInert rule). For example, calls to objects which are not part of a session are allowed at all times. We ensure that the additionally allowed statements do not interfere with a session or its progress.

Therefore we claim that the following theorem holds for our type system:

Theorem 3.4.1 (Method-Local Soundness of our Type System)**.** Let G be a global session type. Let M be the set of methods mentioned in G. Let *Model* be an ABS model. If *Model* is typed by G, that is "*Model* : G" as by the type system of Figures 3.21 to 3.23, then the implementation of every method $m \in M$ in *Model* locally complies with G. This means, for every $p \xrightarrow{f} q\colon m$ in G, for every possible event history h of *Model*, the history $(h \restriction q) \restriction f$ is future-equivalent to a history that fits the regular expression $\tau(\mathit{project}^q_f(\mathit{project}^q(G)))$, assuming m is activated and always reactivated for all suspending statements and no exceptions are thrown.

See also the formal semantics of global types in Section 2.3.1.

Since activations of methods can interleave in any way in an ABS model which does not employ User-defined Schedulers, our type system alone can not guarantee soundness on a global scale. We extend ABS models with schedulers that enforce the (re-)activation order as specified in G in Section 3.5. This results in the global soundness theorem 3.5.1, see Section 3.5.8.

3.5. Dynamic Enforcement

We want to achieve the following objectives by influencing the run-time scheduling behavior of an ABS model:

1. **Enforcement of Specified Execution Order.** Messages, that is calls, can arrive in any order due to the concurrency model of ABS, see Section 2.1.3. Also, without a scheduling function, (re-)activations of methods are scheduled in an undefined order, see Section 2.2.4. Therefore, the static verification of session types can not fully guarantee the order of method activations and we enforce it at runtime instead.

2. **A Basic Safety Guarantee.** Methods and classes which have not been inspected during static verification may interfere with a session in a model which is not closed. We want to give some basic safety guarantee for this case.

Objective 2 is discussed in Section 3.6. The following example illustrates objective 1:

Example 4: Execution Order Violation

The following local session type for an active object *ListView* specifies that first some data must be set and then a user interface is updated to display it:

$$\mathit{Controller}?_{f}\mathrm{setData}.\,\mathrm{Put}\ f.\mathit{Controller}?_{f'}\mathrm{updateUI}.\,\mathrm{Put}\ f'$$

The class `ListView` strives to implement the above type, whereas `Controller` issues the calls being received by it:

```
class ListView
  implements UI {

  Unit setData(Data d)
    { ... }
  Unit updateUI()
    { ... }
}
```

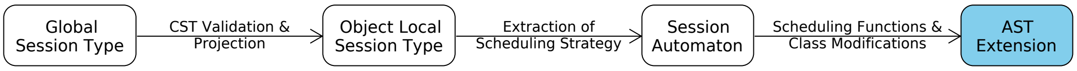
```
class Controller (UI ui)
  implements ControllerI {

  Unit displayData() {
    ui!setData(d);
    ui!updateUI();
  }
}
```

Since method `displayData()` of the `Controller` class does not block to await the completion of the data message, the following execution orders for an instance of `ListView` are possible:

$$\begin{gathered}\mathrm{setData}, \mathrm{updateUI}\\ \mathrm{updateUI}, \mathrm{setData}\end{gathered}$$

The second order violates the session type specification.

We approach these issues by enforcing a *scheduling policy* at runtime which locally ensures the order of invocations for an active object as described by a local session type. First, a description of the *scheduling policy* as a *Session Automaton* is derived from the session type. Session Automata are a subclass of Register Automata [31] and have been introduced by Bollig et al. [8]. Kamburjan et al. developed a method to derive scheduling policies represented as Session Automata from session types in [30]. We describe the construction of an automaton in slightly more detail, see Section 3.5.1, and implemented it as part of the SDS-tool. Next, the AST of all classes implementing actors of a session type is extended, so that it integrates such an automaton and makes scheduling decisions based on the automaton's state, see Section 3.5.6.


Global Session Type
CST Validation & Projection
Object Local Session Type
Extraction of Scheduling Strategy
Session Automaton
Scheduling Functions & Class Modifications
AST Extension

Extending the AST of classes so that they can enforce our scheduling policies is unfortunately not possible in the current ABS implementation. Therefore, we apply a set of modifications to the ABS compiler, see Section 3.5.5. There is also a class of session types for which activation order can not be reliably enforced using schedulers. We review this issue in Section 3.5.7. Finally, using our scheduler extensions, we extend the soundness theorem 3.4.5 in Section 3.5.8.

3.5.1. Session Automata

First lets define our variation of Session Automata based on [8] and [30]:

Definition 3.5.1 (k-register Session Automaton)**.** A Session Automaton is a tuple $A = (Q, q_0, \Delta, F)$. Q denotes the finite set of states, $q_0 \in Q$ the initial state and $F \subseteq Q$ the set of final states. We define the alphabet of labels as

$$\Sigma = \{\mathit{InvocREv}(m), \mathit{ReactEv}(m) \mid m \in \mathit{Methods}\}$$

We denote $\Delta \subset Q \times (\Sigma \times R) \times Q$ as the finite set of transitions. R is the finite set of registers where $|R| = k$.

If we treat a Session Automaton A as a nondeterministic finite automaton (NFA), then we call its language $L_{\mathit{symb}}(A) \subseteq (\Sigma \times R)^*$ the *symbolic language* of A. Its elements are called *symbolic words*. The *data language* $L_{\mathit{data}}(A) \subseteq (\Sigma \times \overline{\mathcal{F}})^*$ of A is the set of *data words* on which there exists a *run* of A.

Definition 3.5.2 (Runs of Session Automata)**.** Let $I = \{0 \ldots n\}$ and $I^+ = I \cup \{n+1\}$. Let $S = R \to (\{\varepsilon\} \cup \overline{\mathcal{F}})$ be the set of *data stores* which map a register to the future stored in it or nothing (ε) if it is empty. For a Session Automaton $A = (Q, q_0, \Delta, F)$ a sequence $(q_i, \sigma_i)_{i \in I^+} \in (Q \times S)^{n+2}$ is a *run* of A for a *data word* $(v_i, f_i)_{i \in I}$ iff the following holds:

$\sigma_0 = R \times \{\varepsilon\}$ and for all $i \in I$ there exists a method m and a register r so that
either $v_i = \mathit{InvocREv}(m)$ and $(q_i, (\mathit{InvocREv}(m), r), q_{i+1}) \in \Delta$ and $\sigma_{i+1} = \sigma_i[r := f_i]$
or $v_i = \mathit{ReactEv}(m)$ and $(q_i, (\mathit{ReactEv}(m), r), q_{i+1}) \in \Delta$ and $\sigma_{i+1} = \sigma_i$ and $\sigma_i(r) = f_i$

Intuitive Explanation

Every state of a Session Automaton A can be interpreted as a progression point of a session. The transitions which are possible in this state model which method activations are allowed at this point in the session. A label $v = \mathit{InvocREv}(m)$ denotes that a method m is being invoked. A label $v = \mathit{ReactEv}(m)$ denotes that a method m which has been suspended is reactivated. The data word character (v, f) then models that the activation denoted by v is represented by future f. A transition $(q_1, (\mathit{InvocREv}(m), r), q_2)$ models that in state q_1 the scheduling policy described by A allows an activation $(\mathit{InvocREv}(m), f)$ resulting in state q_2. Also, if this activation is scheduled, f shall be stored in register r. Analogously, a transition $(q_1, (\mathit{ReactEv}(m), r), q_2)$ models that the policy allows in state q_1 to transition into q_2 when reading a reactivation $(\mathit{ReactEv}(m), f)$. However, this is only permitted if f is the same future as the one stored in register r.

We can construct a scheduler from a Session Automaton so that it implements the scheduling policy described by it. This is achieved by simulating the automaton as part of the scheduler. The scheduler takes activations which are ready to execute as input and schedules one of them which matches a transition of the automaton's current state. If none match a transition, then activation of a process is delayed until a matching one is available.

Since we believe it supports a better understanding of the behavior of schedulers in our examples, we also give a model of such a scheduler. Its states are the elements of a run and the sequences it reads are data words.

Definition 3.5.3 (Session Scheduler)**.** For a Session Automaton $\mathcal{A} = (Q, q_0, \Delta, F)$, we define its *Session Scheduler* as a transition system $(\hat{Q}, \hat{q_0}, \hat{\Delta})$ where $\hat{Q} = Q \times S$ is the set of states and $\hat{q_0} = (q_0, R \times \{\varepsilon\})$ is its initial state. We define

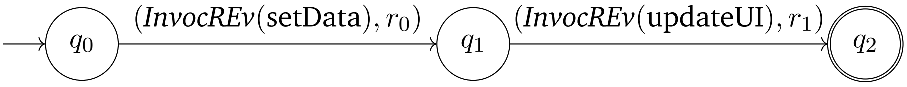
$$
\begin{aligned}
&\hat{\Delta} \subseteq \hat{Q} \times \Sigma \times \overline{\mathcal{F}} \times \hat{Q}\\
&\hat{\Delta} = \{((q, \sigma), v, f, (q', \sigma')) \mid (q, (v, r), q') \in \Delta\\
&\quad \wedge (\\
&\qquad\quad v = \mathit{InvocREv}(m) \wedge \sigma' = \sigma[r := f]\\
&\qquad \vee\, v = \mathit{ReactEv}(m) \wedge \sigma(r) = f \wedge \sigma' = \sigma\\
&\quad)\\
&\}
\end{aligned}
$$

as the system's transition relation.

The following example demonstrates how the concept of Session Automata can be used to enforce the execution order of example 4:

Example 5: Invocation Reordering Via Automata

The following session automaton A encodes a *scheduling policy* which enforces the call order as specified in the session type of example 4:


q_0
$(\mathit{InvocREv}(\text{setData}), r_0)$
q_1
$(\mathit{InvocREv}(\text{updateUI}), r_1)$
q_2

Now, lets illustrate the behavior of the *scheduling policy* described by A by simulating a run using its Session Scheduler where the invocations of setData and updateUI arrive in the wrong order at *ListView*:

#	State	Available activations	Resulting State
1	$(q_0, \{(r_0, \varepsilon), (r_1, \varepsilon)\})$	–	no change
2	$(q_0, \{(r_0, \varepsilon), (r_1, \varepsilon)\})$	$(\mathit{InvocREv}(\text{updateUI}), f)$	no change
3	$(q_0, \{(r_0, \varepsilon), (r_1, \varepsilon)\})$	$(\mathit{InvocREv}(\text{updateUI}), f)$, $(\mathit{InvocREv}(\text{setData}), f')$	$(q_1, \{(r_0, f'), (r_1, \varepsilon)\})$
4	$(q_1, \{(r_0, f'), (r_1, \varepsilon)\})$	$(\mathit{InvocREv}(\text{updateUI}), f)$	$(q_2, \{(r_0, f'), (r_1, f)\})$
5	$(q_2, \{(r_0, f'), (r_1, f)\})$	–	no change

In line 2, only the method updateUI can be activated but this is not permitted by the session automaton in the initial state. Thus, there is no viable transition in the Session Scheduler. However, as soon as an activation of method setData is available in line 3, the system can progress.

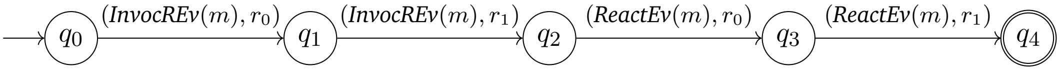
The next example illustrates why we need registers to remember the future of an invocation so that we can discern reactivations of the same method:

Example

Let a be the actor whose behavior is specified by the following object local type:

$$\begin{aligned}
&p?_{f_0} m. \\
&\quad q!_{f_{q1}} m'.\mathit{Await}(f_0, f_{q1}). \\
&p?_{f_1} m. \\
&\quad q!_{f_{q2}} m'.\mathit{Await}(f_1, f_{q2}). \\
&\text{React } f_0.\, \text{Put } f_0. \\
&\text{React } f_1.\, \text{Put } f_1
\end{aligned}$$

The type expresses that a receives two calls from actor p on method m. Both times, a calls method m' on actor q and waits for the result. The type also demands, that after q completes a call, the computations of a still complete in order of their initial invocation. This Session Automaton A describes a *scheduling policy* for a which allows to enforce this order:


q_0
$(\mathit{InvocREv}(m), r_0)$
q_1
$(\mathit{InvocREv}(m), r_1)$
q_2
$(\mathit{ReactEv}(m), r_0)$
q_3
$(\mathit{ReactEv}(m), r_1)$
q_4

In following run of A's Session Scheduler, the second call to q finishes before the first one. This makes the reactivation of the computation denoted by future symbol f_1 possible before the reactivation of f_0. Nevertheless, the scheduler compares the future of the reactivations against the ones stored in its registers. This way it is able to discern the two reactivations. Consequently, it delays the reactivation of f_1 in line 3 and waits for f_0 to complete instead:

#	State	Available activations	Resulting State
1	$(q_0, \{(r_0, \varepsilon), (r_1, \varepsilon)\})$	$(\mathit{InvocREv}(m), f_0)$	$(q_1, \{(r_0, f_0), (r_1, \varepsilon)\})$
2	$(q_1, \{(r_0, f_0), (r_1, \varepsilon)\})$	$(\mathit{InvocREv}(m), f_1)$	$(q_2, \{(r_0, f_0), (r_1, f_1)\})$
3	$(q_2, \{(r_0, f_0), (r_1, f_1)\})$	$(\mathit{ReactEv}(m), f_1)$	no change
4	$(q_2, \{(r_0, f_0), (r_1, f_1)\})$	$(\mathit{ReactEv}(m), f_1)$, $(\mathit{ReactEv}(m), f_0)$	$(q_3, \{(r_0, f_0), (r_1, f_1)\})$
5	$(q_3, \{(r_0, f_0), (r_1, f_1)\})$	$(\mathit{ReactEv}(m), f_1)$	$(q_4, \{(r_0, f_0), (r_1, f_1)\})$
6	$(q_4, \{(r_0, f_0), (r_1, f_1)\})$	–	no change

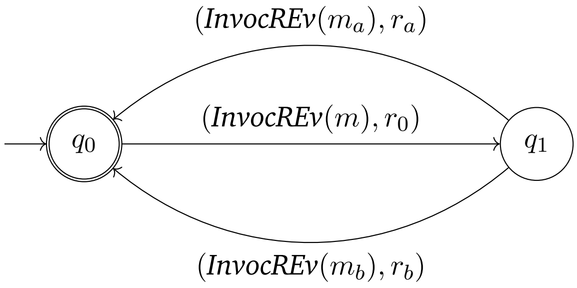
Our last example in this subsection shows why a simple sequential representation instead of an automaton is not sufficient to describe *scheduling policies*. Automata allow us to describe repeating and branching session types through circular paths and multiple possible transitions per state:

Example

Imagine the following object local type with branchings and a repetition:

$$\left(p?_{f_0}m.\,\mathrm{Put}\ f_0.\&_f \left\{\begin{array}{l} p?_{f_a}m_a.\,\mathrm{Put}\ f_a, \\ p?_{f_b}m_b.\,\mathrm{Put}\ f_b \end{array}\right\}\right)^*$$

This Session Automaton describes a *scheduling policy* which enforces the call order as specified by the type:


$(\mathit{InvocREv}(m_a), r_a)$
$(\mathit{InvocREv}(m), r_0)$
q_0
q_1
$(\mathit{InvocREv}(m_b), r_b)$

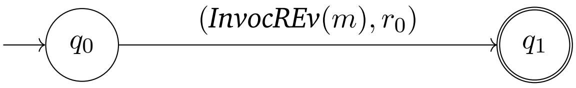
3.5.2. Automaton Generation

Function *genAutomaton* of Figures 3.26 to 3.28 formalizes the generation of a Session Automaton from an object local type. The resulting automata describe a scheduling policy which enforces the invocation and reactivation order as specified by the type. Our formalization carries out the construction principles as outlined by [30], though we eliminate ε-transitions after the construction and transform the resulting automaton into a so-called *symbolically deterministic* automaton, see Section 3.5.3. ε-Transitions simplify the construction process, but automata without ε-transitions on the other hand are more straightforward to transform into AST-extensions. That is why we extend the structure of transitions Δ to allow for ε-transitions during the application of *genAutomaton*:

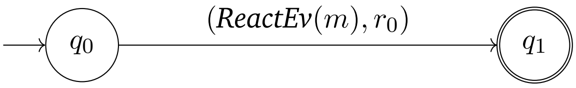
$$\Delta \subseteq (Q \times \Sigma \times R \times Q) \cup (Q \times \{\varepsilon\} \times Q)$$

The following list gives an overview of the construction principles. Below it is a more detailed explanation of the formalization.

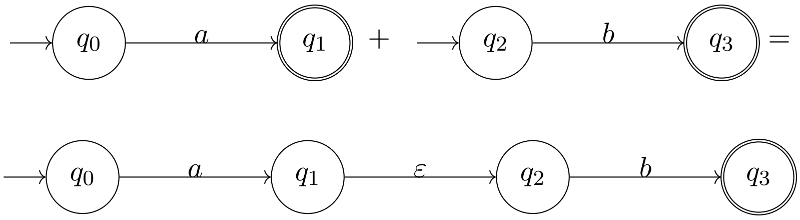
Case 1 Receiving types $\boldsymbol{p?_f m}$ result in an automaton with two states. It allows a transition from the initial state to the second, final state when an invocation of m can be activated. The automaton describes a *scheduling policy* which allows to schedule a single invocation of method m.



Case 2 Reactivation types **React** $\boldsymbol{f}$ also result in an automaton with two states. It also has a single transition from the initial to the second, final state. The resulting automaton describes a policy where a single reactivation is allowed. Only the future in the register where f was stored at its first activation may be reactivated.

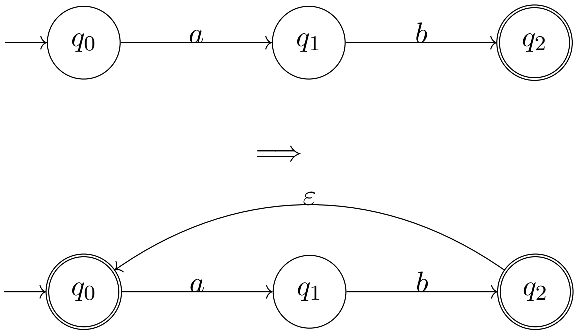


Case 3 Automata for concatenated types $\boldsymbol{L_1.L_2}$ are constructed by concatenation of the automata generated from L_1 and L_2. The described *scheduling policy* follows first the behavior specified in L_1 and then the behavior of L_2.

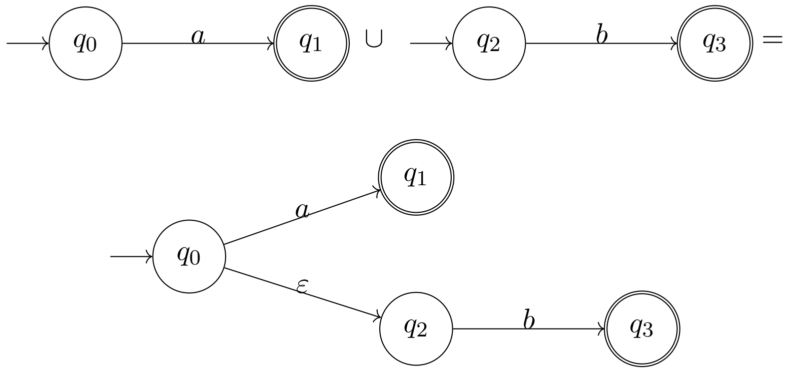


Case 4 For repetitions $(\boldsymbol{L})^*$, first the automaton of L is constructed. Then, cycles are introduced by adding ε-transitions from the final states to the initial state. Since a repetition may not take place at all, the initial state is also made a final state.

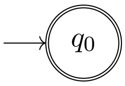
The produced automaton describes a *scheduling policy* according to the behavior described in L but also allows this behavior to repeat due to the cycles.



Case 5 The automata for the branching types $\oplus\left\{(\boldsymbol{L_i})_{\boldsymbol{i\in I}}\right\}$ and $\boldsymbol{\&_f}\left\{(\boldsymbol{L_i})_{\boldsymbol{i\in I}}\right\}$ are composed by generating automata for every branch and computing their union. These automata encode the *scheduling policies* induced by types $(L_i)_{i\in I}$. The policy branches, whenever the behavior specified in the types diverges.



Case 6 For all other types, an automaton with a single, final state and no transitions is generated. See Equation (3.53).



Construction Details

We denote the set of all Session Automata states by $\mathcal{Q}$ and the set of all Session Automata by $\mathcal{A}$. The *genAutomaton* function takes an object local type, a set of used states Q_{used} and a future-method-register mapping R_{map} as input, see formula 3.50. It outputs a Session Automaton and an updated future-method-register mapping. The object local type parameter is the type from which a Session Automaton shall be derived. For some types, we must construct sub-automata and recombine them. This is easier to implement if we make sure that their states do not intersect. Therefore, the only cases where new states are needed, which are Equations (3.51) to (3.53), ensure that no states from Q_{used} are used for construction. Consequently, the other cases, e. g. Equation (3.55), pass on the set of used states between recursive uses of *genAutomaton*. The same reasoning applies to the mapping R_{map} and the register names it stores. Equation (3.51) notes in R_{map} which method was used to produce a future and in which register it is stored by the corresponding *InvocREv* transition. This information is required by Equation (3.52) to construct *ReactEv* transitions.

We now examine the formalization for the individual cases:

Case 1 – Eqn. 3.51 Given a receiving type $p?_f m$, an automaton is constructed from two new states q_0 and q_1. They are connected by a single *InvocREv* transition for the invoked method as specified in the session type. q_0 is the initial and q_1 a final state.

Case 2 – Eqn. 3.52 This case is very similar to **Case 1**. The only difference is, that for a type React f a *ReactEv* transition is encoded.

Case 3 – Eqn. 3.55 For a concatenated type $L_1.L_2$ automata A_1 and A_2 are constructed for L_1 and L_2 respectively and then concatenated. Concatenation is defined as function concatAutomata in Figure 3.29. It works by inserting a ε-transition to the initial state of A_2 outgoing from a set of states of A_1 we call Q_{glue}. Also, the final states of the resulting automaton are replaced with a set F. In this case, the outgoing states are the final states F_a of the first automaton $Q_{glue} = F_a$ and the new final states will be the final states F_b of the second automaton. We added the parameters Q_{glue} and F to the function concatAutomata since this enables us to implement **Case 5** also via a variation of concatenation.

Case 4 – Eqn. 3.54 For repeated types $(L)^*$ an automaton is constructed from L first. Then ε-transitions are added from its final states q_f to its initial state q_0, forming cycles. q_0 is also added as a final state.

Case 5 – Eqn. 3.56 For branching types $\oplus\left\{L_1, L_2, \ldots L_n\right\}$ and $\oplus\left\{L_1, L_2, \ldots L_n\right\}$ a "*head*"-automaton is constructed from the first branch L_1 and a "*tail*"-automaton for the branching type $\oplus\left\{L_2, L_3, \ldots L_n\right\}$ of the $n-1$ remaining branches. The base case of this recursion on a branching type is the case where no branches are left $(n = 0)$ and its handled by **Case 6**. Therefore, Equation (3.56) has the requirement $n > 0$.

Next, the union of the "*head*"- and "*tail*" automata is computed by a variation of concatenation. This time, the ε-transitions originate from the initial state of the "*head*"-Automaton ($Q_{glue} = \{q_{head;0}\}$). Also, we keep all final states ($F = F_{head} \cup F_{tail}$).

$$\begin{aligned}\mathfrak{R}_{map} &= 2^{\mathcal{F}\times \mathit{Methods}\times R}\\ \mathit{genAutomaton}\colon \mathcal{L}\times 2^{\mathcal{Q}} \times \mathfrak{R}_{map} &\to \mathcal{A}\times \mathfrak{R}_{map}\end{aligned} \tag{3.50}$$

$$\begin{aligned}&\mathit{genAutomaton}(p?_f m, Q_{used}, R_{map}) =\\ &\quad(\\ &\qquad(\{q_0,q_1\}, q_0, \{(q_0,(\mathit{InvocREv}(m), r), q_1)\}, \{q_1\}),\\ &\qquad R_{map}\cup\{(f,m,r)\}\\ &\quad)\\ &\quad\text{where } q_0,q_1\notin Q_{used}\\ &\quad\text{and } \forall f',m'.\,(f',m',r)\notin R_{map}\end{aligned} \tag{3.51}$$

$$\begin{aligned}&\mathit{genAutomaton}(\text{React}\, f, Q_{used}, R_{map}) =\\ &\quad(\\ &\qquad(\{q_0,q_1\}, q_0, \{(q_0,(\mathit{ReactEv}(m), r), q_1)\}, \{q_1\}),\\ &\qquad R_{map}\\ &\quad)\\ &\quad\text{where } q_0,q_1\notin Q_{used}\\ &\quad\text{and } (f,m,r)\in R_{map}\end{aligned} \tag{3.52}$$

$$\begin{aligned}&\mathit{genAutomaton}(L, Q_{used}, R_{map}) =\\ &\quad(\\ &\qquad(\{q_0\}, q_0, \emptyset, \{q_0\}),\\ &\qquad R_{map}\\ &\quad)\\ &\quad\text{where } q_0\notin Q_{used}\\ &\quad\text{and } L\notin \{\text{React}\, f, p?_f m, \&_f\left\{(L_i)_{i\in I}\right\}, \oplus\left\{(L_i)_{i\in I}\right\}, \left(L'\right)^* \mid\\ &\qquad\quad f\in\mathcal{F}\wedge p\in \mathit{Actors}\wedge m\in\mathit{Methods}\wedge L'\in\mathcal{L}\wedge I\neq\emptyset\wedge\forall i.\, L_i\in\mathcal{L}\}\end{aligned} \tag{3.53}$$

Figure 3.26.: Formalization of the simple automaton generation cases **Case 1**, **Case 2** and **Case 6**.

$$\begin{aligned}
&\mathit{genAutomaton}((L)^*, Q_{used}, R_{map}) = \\
&\quad (\\
&\quad\quad (\\
&\quad\quad\quad Q', \\
&\quad\quad\quad q_0, \\
&\quad\quad\quad \Delta \cup \{(q_f, v, q_1) \mid q_f \in F \wedge (q_0, v, q_1) \in \Delta\}, \\
&\quad\quad\quad F \cup \{q_0\} \\
&\quad\quad), \\
&\quad\quad R'_{map} \\
&\quad) \\
&\quad \text{where } ((Q', q_0, \Delta, F), R'_{map}) = \mathit{genAutomaton}(L, Q_{used}, R_{map})
\end{aligned} \tag{3.54}$$

$$\begin{aligned}
&\mathit{genAutomaton}(L_1.L_2, Q_{used}, R_{map}) = \\
&\quad (\\
&\quad\quad \text{concatAutomata}(A_1, A_2, F_1, F_2), \\
&\quad\quad R''_{map} \\
&\quad) \\
&\quad \text{where } (A_1, R'_{map}) = ((Q_1, \ldots, F_1), R'_{map}) = \mathit{genAutomaton}(L_1, Q_{used}, R_{map}) \\
&\quad \text{and } (A_2, R''_{map}) = ((\ldots, F_2), R''_{map}) = \mathit{genAutomaton}(L_2, Q_1, R'_{map})
\end{aligned} \tag{3.55}$$

Figure 3.27.: Formalization of the automaton generation cases **Case 4** and **Case 3**.

$$
\begin{aligned}
&\mathit{genAutomaton}((\cdot)\left\{L_1, L_2, \dots L_n\right\}, Q_{\mathit{used}}, R_{\mathit{map}}) = \\
&\quad (\\
&\quad\quad \text{concatAutomata}(A_{\mathit{head}}, A_{\mathit{tail}}, \{q_{\mathit{head};0}\}, F_{\mathit{head}} \cup F_{\mathit{tail}}) \\
&\quad\quad R''_{\mathit{map}} \\
&\quad) \\
&\text{where } (A_{\mathit{head}}, R'_{\mathit{map}}) = ((Q_{\mathit{head}}, q_{\mathit{head};0}, \Delta_{\mathit{head}}, F_{\mathit{head}}), R'_{\mathit{map}}) = \mathit{genAutomaton}(L_1, Q_{\mathit{used}}, R_{\mathit{map}}) \\
&\text{and } (A_{\mathit{tail}}, R''_{\mathit{map}}) = ((\dots, F_{\mathit{tail}}), R''_{\mathit{map}}) = \mathit{genAutomaton}((\cdot)\left\{L_2, L_3, \dots L_n\right\}, Q_{\mathit{head}}, R'_{\mathit{map}}) \\
&\text{and } n > 0
\end{aligned}
\tag{3.56}
$$

Figure 3.28.: Function for generating a Session Automaton from a branching session type. $(\cdot)\{(L_i)_{i\in I}\}$ is to be replaced with either $\oplus\left\{(L_i)_{i\in I}\right\}$ or $\&_f\left\{(L_i)_{i\in I}\right\}$ for the whole definition.

$$
\begin{aligned}
&\text{concatAutomata}((Q_a, q_{a;0}, \Delta_a, F_a), (Q_b, q_{b;0}, \Delta_b, F_b), Q_{\mathit{glue}}, F) = \\
&\quad (\\
&\quad\quad Q_a \cup Q_b, \\
&\quad\quad q_{a;0}, \\
&\quad\quad \Delta_a \cup \Delta_b \cup \Delta_\varepsilon, \\
&\quad\quad F \\
&\quad) \\
&\text{where } \Delta_\varepsilon = \{(q_g, \varepsilon, q_{b;0}) \mid q_g \in Q_{\mathit{glue}}\}
\end{aligned}
$$

Figure 3.29.: Concatenation function for Session Automata using ε-transitions.

3.5.3. Transformation into Symbolically Deterministic Automata

For Session Automata, Bollig et al. differentiate two kinds of determinism [8]. In short, a Session Automaton A is *symbolically deterministic*, if in every state there is at most one transition for a symbolic word character. It is the same notion of determinism as if we would regard A as a finite automaton. A is *data deterministic* if it is symbolically deterministic and for every point of a run sequence, there is at most one possible successor for any data word character.

Ideally, since we want to implement schedulers as an extension of an ABS model, we want to track at most one possible automaton state and avoid nondeterminism. This simplifies the implementation and is also more memory efficient. Tracking just one state is possible if our automata were data deterministic. However, we are only partially able to achieve this property for our automata.

Transforming a generated automaton into a symbolically deterministic one is easy: If two automata have the same symbolic language, they also have the same data language [8]. Therefore, we can use a variant of the powerset constuction [22] to transform a generated automaton into a symbolically deterministic one while also removing ε-transitions, see Algorithm 2. ε-Transitions are eliminated by replacing states with the closure of states reachable by ε-Transitions [23].

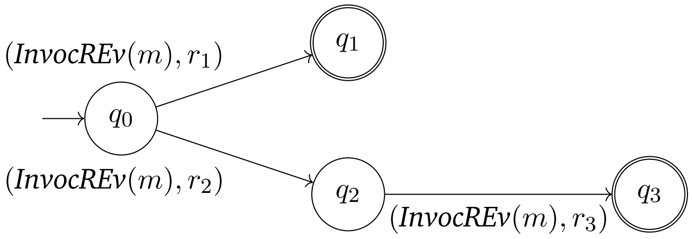
Our automata are constructed in such a way that every distinct future is stored at most once in one register. Thus, our symbolically deterministic automata are also data deterministic in regard to *ReactEv* transitions, since in no run there are two registers storing the same future.

However, there can still be data nondeterminism regarding *InvocREv* transitions.

Example 6: Data Nondeterminism in Symbolically Deterministic Automata
Imagine the following object local session type:

$$\&_f \begin{Bmatrix} p?_{f_a}m, \\ p?_{f_b}m.p?_{f_b'}m \end{Bmatrix}$$

The automaton A has been generated from it:


$(\mathit{InvocREv}(m), r_1)$
q_1
q_0
$(\mathit{InvocREv}(m), r_2)$
q_2
$(\mathit{InvocREv}(m), r_3)$
q_3

The powerset construction for finite automata does not merge the two invocation transitions of state q_0. When encountering the data word character $(\mathit{InvocREv}(m), f)$ a scheduler can not know, which branch of the specification the model execution entered, since there are two fitting transitions. If it was simulating A as a nondeterministic automaton, it could apply both transitions and keep track of the two possible states q_1 and q_2 and the possible data stores associated with them. If later an activation $(\mathit{InvocREv}(m), f')$ became available, it could still schedule it, since the transition is allowed in q_2.

This is not possible, if a scheduler was only able to keep track of one automaton state.

We decided to simulate just a single automaton state and data store in our AST extension due to the benefits mentioned above and also because data-nondeterminism occurs only in the case where multiple branches of a Session Type begin with the same method call. Instead, we detect data-nondeterminism by checking, whether there is any state in an automaton (Q, q_0, Δ, F) where two or more *InvocREv* transitions target the same method:

$$\exists q_1 \in Q. \left|\{q_2 \mid (q_1, \mathit{InvocREv}(m), r, q_2) \in \Delta\}\right| > 1$$

If such a case is detected, our AST extension issues a warning at runtime.

Please note that a full transformation into a data deterministic Session Automaton is likely possible by adapting Algorithm 2 so that it merges *InvocREv* transitions of the same state when targeting the same method. However, this would require a proof that the transformed automaton still accepts the same data language, especially since data deterministic automata are less expressive than symbolically deterministic ones [8]. Due to the time constraints on this thesis we leave this improvement open as future work.

3.5.4. Summary of Automaton Generation Steps

For a given actor p and global session type G we denote a Session Automaton generated from G for p by $\mathfrak{A}(G, p)$. It is constructed by the following steps:

1. G is projected on p to the local type L.
2. A Session Automaton A_ε with ε-transitions is constructed from L using the function *genAutomaton*.
3. ε-Transitions and symbolic-nondeterminism are removed using Algorithm 2.

Therefore

$$
\begin{aligned}
&\mathfrak{A}(G, p) = \varepsilon\text{NFA}\textsc{To}\text{DFA}(\\
&\quad \textit{genAutomaton}(\\
&\qquad \textit{project}^p(G), \emptyset, \emptyset\\
&\quad)\\
&)
\end{aligned}
$$

3.5.5. ABS Compiler Modifications

Our goal is to influence the scheduling behavior of ABS classes implementing actors so that it adheres to a session type. The only method to influence scheduling behavior of a COG in ABS are scheduling functions (Section 2.2.4). Thus, we use scheduling functions to decide which activations to schedule depending on the state of a simulated Session Scheduler. However, the currently implemented *User-defined Scheduling Functions* are not sufficient for our purposes. Therefore, we had to apply the following modifications to the ABS compiler:

1. make the return value of scheduling functions optional.
2. implement the **`destiny`** expression of ABS Core.
3. introduce the `Any` type at the top of the type hierarchy.

These modifications are explained in detail in the following sections.

Compiler Modification 1: Scheduling Functions with Optional Return Value

Since there are cases where none of the available activations are viable for scheduling (see Example 5), scheduling functions must be able to delay activations until a viable one is available.

This is not possible with the current implementation of scheduling functions which requires them to have a return value of type `Process`. We modify the *abstools* compiler and Erlang backend, so that scheduling functions require the return type `Maybe<Process>`. The type has two constructors:

`Nothing`
: Returning a value created with this constructor signifies that no activation is viable. Thus, scheduling an activation is delayed by the backend.

`Just(Process fromJust)`
: If a value `Just(p)` is returned, the activation represented by `p` is scheduled by the backend.

Compiler Modification 2: `destiny` Expression

To be able to store a future which a method is currently computing in a register, there must be a way to access it. Though Core ABS defines an expression **`destiny`** which represents this future [27], it is not implemented by *abstools*. Moreover, scheduling functions must be able to access the future of an activation from a process value `p` to be able to compare it to registers. However, this is also not possible.

Therefore, we extended the *abstools* compiler and implemented the **`destiny`**-expression and a function `destinyOf` which returns the future of a given `Process` value.

Compiler Modification 3: `Any` Type

Future types in ABS carry the type of their result as a type parameter, e. g. `Fut<Int>`. However, the `Process` type representing activations makes it impossible to define a concrete return type for the `destinyOf` function that we introduced in the above section:

```
def Fut<???> destinyOf(Process p) = builtin;
```

We solved this issue by adding a new built-in type Any at the top of the subtyping hierarchy of ABS, so that

$$T \preceq \texttt{Any} \land \texttt{Any} \preceq \texttt{Any}$$
$$\text{for all classes and interfaces } T$$

where $\preceq$ is the subtyping relation of ABS. Moreover, a flat subtyping hierarchy is introduced for data types D so that

$$D \preceq_{\mathcal{D}} \texttt{Any} \land \texttt{Any} \preceq_{\mathcal{D}} \texttt{Any}$$

where $\preceq_{\mathcal{D}}$ is the subtyping relation of this hierarchy. Though it does not allow other operations, a value of type Any supports equality comparisons. Hence, we can now define the return type of destinyOf as Fut<Any> which still allows us to compare the futures of possible activations against registers:

```
def Fut<Any> destinyOf(Process p) = builtin;
```

Please note, that of course other solutions to this problem are conceivable which would not require to modify the type hierarchy. For example, a built-in custom comparison operator between Process and Fut<T>. However, such a solution would also require us to modify the ABS type system to not check this operation. Also, Any enables us to define a register type which can store any future type. This slightly simplifies the generation of automaton AST extensions in the following section.

3.5.6. Automaton Integration

Now that we have Session Automata to describe *scheduling policies* and have extended the ABS compiler sufficiently, we need a method to modify ABS classes so that they schedule activations as modeled by an automaton's Session Scheduler. This is achieved by extending classes to simulate their Session Automaton.

Though a clean separation of a class implementing an actor and its scheduling extension is desirable from a software engineering perspective, it is not possible due to these limitations:

- scheduling functions are pure, therefore changes to the scheduler's state must be performed by the scheduled methods.

- the "traits"-feature of ABS allows to modify methods and separate the modification from a class. It can only insert statements preceding and following the original method. However, to implement *ReactEv* transitions, we need to modify the scheduler's state after every **await**-statement, which is not possible with "traits".
- state can not be stored as part of the scheduling function. Hence, classes implementing an actor must be extended with fields storing the state of the simulated Session Scheduler.

The state $\hat{q} \in Q \times S$ of a Session Scheduler consists of the state $q \in Q$ of the automaton A it has been derived from and a data store $\sigma \in S$ of registers. Since Q is finite and we convert automata into (almost) data-deterministic Session Automata after their generation, we can represent q in ABS as a single integer field. We denote the injective mapping from states Q to integers by *toInt* $: Q \to \mathbb{Z}$. As the number of registers R is also finite, we can represent each register as a field storing either a future or initially a placeholder value (`Nothing`).

From the above considerations, given a Session Automaton $A = (Q, q_0, \Delta, F)$, we derive the following concept for simulating a Session Scheduler as an extension of a class `C`:

Additional Fields The class `C` is extended with a field `Int q =` *toInt*(q_0)`;` of integer type representing the automaton state. Its initial value is the integer representation of the initial automaton state q_0. Also fields

$$(\texttt{Register } r \texttt{ = Nothing;})_{r \in R}$$

are added, where $R = \{r | (\cdot, (\cdot, \cdot, r), \cdot) \in \Delta\}$. For abbreviation, we write `Register` for the type `Maybe<Fut<Any>>`.

These fields represent the scheduler's registers. Initially they store the value `Nothing` which signifies that a register has not yet been assigned a future. A future f is assigned to a register by storing the value `Just(`f`)`.

Generation of a Scheduling Function A scheduling function `schedule`[3]is generated which takes a list of possible activations and the current state of the Session Scheduler, which are the above fields, as input:

[3] Of course we use a unique method name for every scheduling function in our tool implementation to avoid name conflicts.

```
def Maybe<Process> schedule(
  List<Process> queue,
  Int q,
  (Register $r)_{r \in R}$
) = forceInit(() => $\langle body \rangle$)(queue);
```

Although we forbid the definition of a custom init-block during static verification (see Section 3.4), all classes implicitly execute a default init-Block upon instantiation. Therefore, we need to ensure, that the generated `schedule` function executes the init-block first, regardless of the automaton's transitions. This is implemented in a second-order function `forceInit`. If there is no init-block to activate, it evaluates and returns $\langle \mathit{body} \rangle$ instead.

For the $\langle \mathit{body} \rangle$ expression, we generate a **`case`**-expression that decides which activation to choose based on the transitions possible in the current scheduler state `q`:

```
$\langle body \rangle =$ case q {
  ($toInt(q')$ => $\langle case\ q' \rangle$;$)_{q' \in Q}$
}
```

This is achieved by filtering the available activations for two criteria:

Either an activation p is a new invocation (its future is not stored in any register) and there is an *InvocREv* transition with the same method name as label:

```
$\langle criterion\ 1 \rangle =$
      !contains(set[$(r)_{r \in R}$], Just(destinyOf(p)))
   && contains(set[$(m)_{m \in invocM}$], method(p))
```

where $\mathit{invocM} = \{m \mid (q', (\mathit{InvocREv}(m), \cdot), \cdot) \in \Delta\}$.

Or the future f of the activation p is stored in a register which is used in a *ReactEv* transition of the current state `q`:

```
$\langle criterion\ 2 \rangle =$
   contains(set[$(r)_{r \in reactR}$], Just(destinyOf(p)))
```

where $\mathit{reactR} = \{r \mid (q', (\mathit{ReactEv}, \cdot, r), \cdot) \in \Delta\}$.

Finally, the first activation from that filtered list is selected, or `Nothing` is returned, if there is no activation which fits the criteria:

```
$\langle \textit{case } q' \rangle =$
  headOrNothing(
    filter((Process p) =>
          $\langle \textit{criterion } 1 \rangle$
       || $\langle \textit{criterion } 2 \rangle$
    )(queue)
  );
```

The class `C` is annotated to use `schedule` as its scheduling function:

```
[Scheduler: schedule(queue, q, $(r)_{r \in R}$)]
class C implements ... { ... }
```

Method Modifications Modifications of the state of the simulated scheduler are achieved by inserting additional statements into the methods of a class. We require two different kinds of modifications for *InvocREv* and *ReactEv* transitions.

***InvocREv* Transitions** First, we group *InvocREv* transitions by the method m that shall be invoked, i. e. the label of a transition is $(\textit{InvocREv}(m), r)$. For each such method m, the AST of the method is prepended with a **`case`**-statement which updates the state of the scheduler. The statement is constructed like this:

1. first the **`case`**-statement itself derives which transition $(q', (\textit{InvocREv}(m), r), q'')$ the scheduling function selected. This can be achieved by inspecting the current state of the scheduler since the method context m is fixed.

   If the scheduler is in a state where m should never be invoked, execution is aborted with an **`assert False`**`;` statement:

   ```
   case this.q {
     $(\textit{toInt}(q')$ => $\langle \textit{case } q' \rangle$;$)_{q' \in Q}$
     _ => assert False;
   }
   ```

2. next, the case statement's branches update the automaton state `q` to $\textit{toInt}(q'')$ and store the future **`destiny`** of the current computation in register r:

```
$\langle \mathit{case}\ q' \rangle$ =
  this.$r$ = destiny;
  this.q = $\mathit{toInt}(q'')$;
  $\langle \mathit{nondeterminismWarning}\ q' \rangle$
```

3. If the the *InvocREv* transition could not be selected in a data deterministic manner (see Section 3.5.3), a warning is printed:

$$\langle \mathit{nondeterminismWarning}\ q' \rangle =$$

$$\begin{cases} \texttt{println("Warning...");} & \text{if } |\{q_2 \mid (q', \mathit{InvocREv}(m), r, q_2) \in \Delta\}| > 1 \\ \texttt{skip;} & \text{otherwise} \end{cases}$$

***ReactEv* Transitions** Next, for every method m of C we compute the set $\Delta_{\mathit{ReactEv}}$ of *ReactEv* transitions of m:

$$\Delta_{\mathit{ReactEv}} = \{\delta \mid \delta = (\cdot, (\mathit{ReactEv}(m), r), \cdot) \in \Delta\}$$

Then the AST of the method is searched for possible reactivation points, which are all positions after **await**-statements. We do not need to consider statements other than **await** statements because the static verification only permits **await** statements for suspension. Directly after these reactivation points, a nested **if**-**else**-statement $\langle \mathit{reactUpdate}(\Delta_{\mathit{ReactEv}}) \rangle$ is inserted:

$$\langle \mathit{reactUpdate}(\Delta') \rangle = \begin{cases} \langle \mathit{ifElse}(\delta, \Delta' \setminus \{\delta\}) \rangle & \text{if} \quad |\Delta'| > 0 \\ & \quad \wedge\, \delta \in \Delta' \\ \texttt{assert False;} & \text{otherwise} \end{cases}$$

where

```
$\langle \mathit{ifElse}((q', (\mathit{ReactEv}, \cdot, r), q''), \Delta'') \rangle$ =
  if (this.q == $\mathit{toInt}(q')$ && this.$r$ == Just(destiny)) {
    this.q = $\mathit{toInt}(q'')$;
  }

  else {
    $\langle \mathit{reactUpdate}(\Delta'') \rangle$
  }
```

$\langle \mathit{reactUpdate}(\Delta_{\mathit{ReactEv}}) \rangle$ compares the current state `this.q` to the starting states of the possible transitions and `destiny` against the register in the transition label to determine which reactivation transition had been selected by `schedule`. If a transition matches, `this.q` is updated accordingly. Otherwise, an impossible state has been encountered and we abort with an assertion.

We give an extensive example illustrating all of the above modifications in the appendix, see Example 8.

Notation Let C be an ABS class, G be a global type and p be an actor in G. Then $\mathfrak{S}(G, p, C)$ denotes the modified AST of C as by the above concept so that it implements the scheduling policy described by $\mathfrak{A}(G, p)$.

3.5.7. Limits of Schedulers

Limitation 1 Our schedulers only differentiate invocations by the name of a method. Therefore, invocations of the same method with different parameter values may be activated in a different order than the calls have been issued by a client. However, the ordering is still consistent with the specification, since our session types also do not encode parameter values.

Example

Given the session type

$$0 \xrightarrow{f} p\colon m'.p \xrightarrow{fa} q\colon m.p \xrightarrow{fb} q\colon m.q \downarrow \mathit{fa}.q \downarrow \mathit{fb}.p \downarrow f$$

and an implementation of m' which contains these calls

```
...
this.fa = q!m(1);
this.fb = q!m(2);
...
```

then the following activation orders are possible for m in q, even if q has been extended with a scheduler derived from the above session type:

`m(1), m(2)` and `m(2), m(1)`

Limitation 2 Only so-called *admissable* types whose communication pattern ensures an activation order of methods can be verified by static methods alone [28]. This is the original motivation schedulers have been applied in [28] to lift this restriction since they can enforce activation orders.

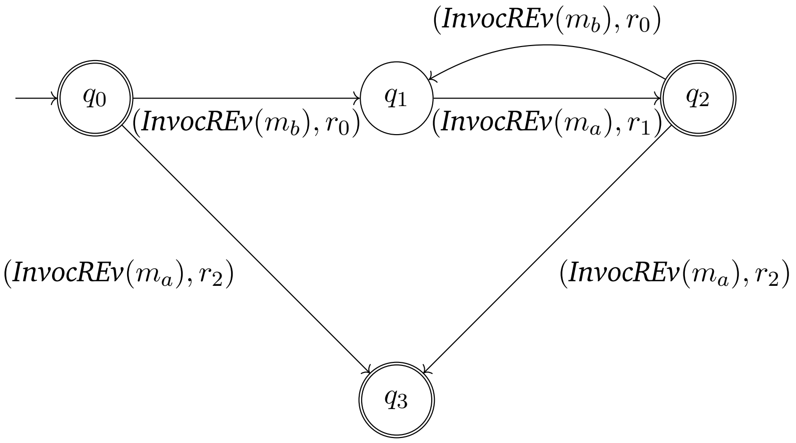
However, because messages can arrive in any order in ABS models, we noticed that there are still some session type specifications where the order of (re-)activations of methods can not be guaranteed by schedulers.

Example

Imagine the following local session type which types an object q:

$$(p?_{f_b}m_b.\,\text{Put}\;f_b.p?_{f_a}m_a.\,\text{Put}\;f_a)^*\;.p?_{f'_a}m_a.\,\text{Put}\;f'_a$$

The following automaton is generated for it:


$(\textit{InvocREv}(m_b), r_0)$
q_0
q_1
q_2
$(\textit{InvocREv}(m_b), r_0)$
$(\textit{InvocREv}(m_a), r_1)$
$(\textit{InvocREv}(m_a), r_2)$
$(\textit{InvocREv}(m_a), r_2)$
q_3

We can fabricate an ABS model where p calls first m_b and then m_a. According to the session type, we should have entered the repeating section of the type and both calls should be executed. However, if the invocation of m_a arrives first at q and the invocation of m_b considerably later, our scheduler implementation executes m_a first. When the invocation of m_b arrives, the simulated automaton will be in state q_3 and m_b will never be executed.

We want to point out this issue but do not pursue a solution in this thesis due to the time constraints it is subject to. Therefore, a precise characterization of this class of session types and an algorithm for detecting them remain as future work. In the meantime, though these types might result in a model execution which can not progress, our schedulers still ensure a basic safety guarantee, see Section 3.6.

3.5.8. Global Soundness

Now that schedulers dynamically enforce the (re-)activation order of methods at runtime, we can complete our soundness theorem of Section 3.4.5. Based on Theorem 3.4.1 and Theorem 2 of [28] we argue that the following theorem holds:

Theorem 3.5.1 (Global Soundness)**.** Let G be a global session type. Let P be the set of all actors in G. Let *Model* be an ABS model. Let $\overline{C}$ be the classes of *Model*. If the following conditions hold

(i) *Model* : G, as by the type system of Figures 3.21 to 3.23

(ii) *Model* is a closed system. That is, every class in $\overline{C}$ performs a role in G. See also Section 3.4.2.

(iii) Every class C in $\overline{C}$ where $p \in P$ is the actor whose role is performed by C has been replaced by $\mathfrak{S}(G, p, C)$ (scheduler modifications).

(iv) All possible executions of *Model* terminate. No exceptions are thrown.

then *Model* complies with G. That is, every possible history of *Model* is “captured” by G.

See also the formal semantics of session types in Section 2.3.1.

Requirement (iv) is necessary because we make only a weak liveness guarantee, see Section 3.6. The implications of models which are not closed are outlined in Section 3.6.

3.6. Liveness and Safety Considerations

By Theorem 3.4.1, our static verification does ensure that methods locally behave like specified in a session type. This gives us the following weak liveness property:

> Let G be a global session type. Let *Model* be an ABS model so that *Model* : G. For all $p \xrightarrow{f} q\colon m$ in G the following holds: If during an execution of *Model* the computation symbolized by f is (re-)activated, then all actions of f as specified in the type are executed up until the next suspending action *Rel*(q, f') or resolving action $q \downarrow f\ [(C)]$.

There are two exceptions to this rule:

1. An ABS exception is thrown. We ignore the possibility of exceptions being thrown during static verification due to the time constraints applying to this thesis. Since our type system forbids **try-catch**-constructs and recovery blocks, exceptions will kill their surrounding object.
2. An assertion fails which will also throw an exception. Since we view assertions as a tool for dynamic verification, our type system permits their use and we also employ them ourselves for dynamic enforcement.

The property necessitates that a method is (re-)activated and whether an activation actually takes place is up to the generated schedulers. These can not strengthen this liveness property due to the following issues:

- we allow for non data-deterministic *InvocREv* transitions, see Section 3.5.3 and Example 6.
- messages can arrive in any order, see Section 3.5.7.
- we allow for half-open systems where an external object may disrupt a session (see subsection below).
- we perform no checks for deadlocks.

Therefore we had to add requirement (iv) to Theorem 3.5.1.

Note regarding Deadlocks Kamburjan et al. also developed a type system for session types in [29] which checks for deadlocks being implied by a session type. Integrating their technique into our tool is an opportunity for future improvement. Alternatively, a framework for detecting deadlocks in Core ABS has also been realized by Giachino et al. in [19].

Half-Open Systems

As made clear in Section 3.4, our static verification only inspects those parts of an ABS model directly referenced by a session type. The only exception to this is a check to ensure, no class implementing an actor is instantiated twice (Section 3.4.4).

This approach implies a half-open system where active objects which are not part of a session type specification might interfere with a session, see Figure 3.30 and Example 7 in the next section.

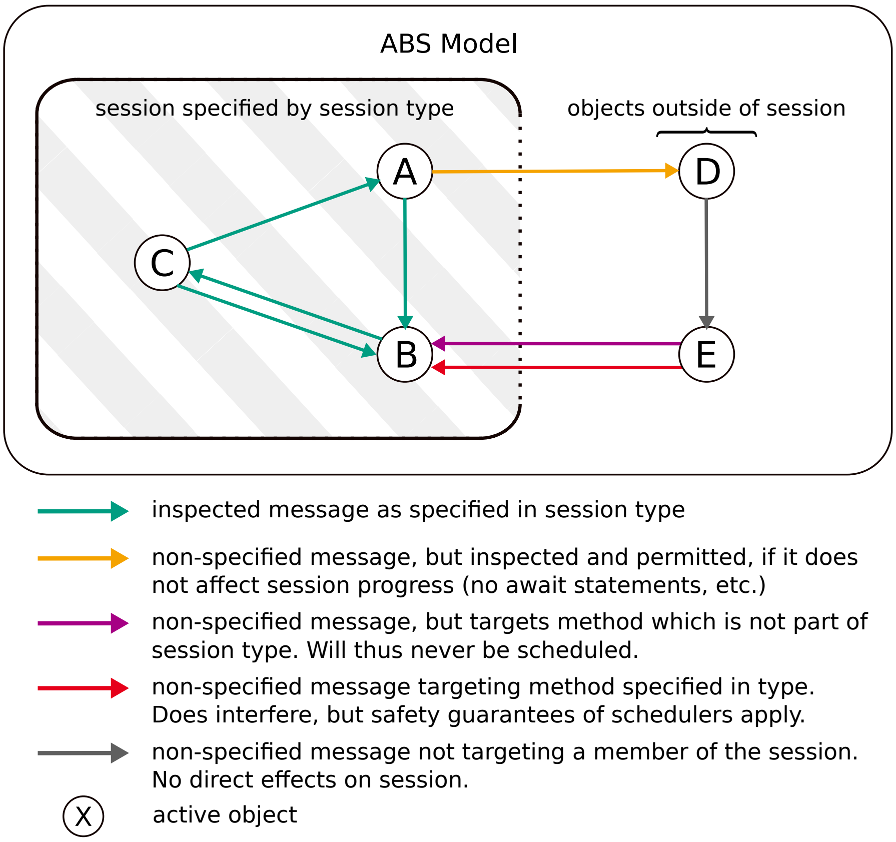

ABS Model
session specified by session type
objects outside of session
A
D
C
B
E
inspected message as specified in session type
non-specified message, but inspected and permitted, if it does not affect session progress (no await statements, etc.)
non-specified message, but targets method which is not part of session type. Will thus never be scheduled.
non-specified message targeting method specified in type. Does interfere, but safety guarantees of schedulers apply.
non-specified message not targeting a member of the session. No direct effects on session.
X
active object

Figure 3.30.: Half-open system as permitted by our static verification process.

Since there is no globally accessible heap in ABS, the only channel for external interference is that some object not participating in a session might call an actor of the session. In this context, we differentiate two kinds of calls:

- In a class which implements an actor of a session, there may be methods that are not inspected during verification because they are not mentioned in the session type. These may influence a session, if executed, for example by modifying fields. However, if called, they are never executed since the automata controlling our schedulers have no *InvocREv* transitions for these methods. Such calls are colored in **purple** in Figure 3.30. See also Section 3.5.2.

- A method whose invocation is permitted in the current state of a used Session Automaton may be called by an object not belonging to the session. Our schedulers do not prevent these invocations but we at least give a basic safety guarantee, see below. Such calls are colored in **red** in Figure 3.30.

We want to point out that the second kind of call could also be prevented by a scheduler if the ABS compiler were to be modified further. Making the name of a caller of a method accessible through `Process` values would suffice.

Safety Guarantee

Our scheduler AST extensions provide the following safety guarantee:

> Let G be a global session type and $(p_i)_{i \in I}$ be actors in G. Let *Model* be an ABS model and $(C'_i)_{i \in I} = (\mathfrak{S}(G, p_i, C_i))_{i \in I}$ be classes of *Model*. Then for all $i \in I$, for all possible executions of *Model*, at most those actions specified in $\mathit{project}^{p_i}(G)$ can be executed in an instance of C'_i and only in the specified order.

The following example clarifies the benefits of this guarantee:

> **Example 7: Safety Guarantees of Schedulers**
> Let there be an ABS model of an operating system. The model encompasses a file `File` which can be opened, data be written multiple times and finally the file must be closed. A user `Alice` who performs these actions is also included in the model. A session type

which specifies the above protocol may look like this:

$$
\begin{aligned}
&0 \xrightarrow{f_0} \texttt{Alice}\colon \texttt{login}.\\
&\texttt{Alice} \xrightarrow{f_{\texttt{open}}} \texttt{File}\colon \texttt{open}.\texttt{File} \downarrow f_{\texttt{open}}.\\
&\left(\texttt{Alice} \xrightarrow{f_{\texttt{write}}} \texttt{File}\colon \texttt{write}.\texttt{File} \downarrow f_{\texttt{write}}\right)^{*}.\\
&\texttt{Alice} \xrightarrow{f_{\texttt{close}}} \texttt{File}\colon \texttt{close}.\texttt{File} \downarrow f_{\texttt{close}}.\\
&\texttt{Alice} \downarrow f_0
\end{aligned}
$$

Now, imagine the model is a half-open system and there is also a user Eve who interacts with File using the open, write and close methods. If File has been extended with a scheduler derived from the above type, then Eve's calls can be executed, however, it is guaranteed that File is never written, if it is not open. Also, if File had a method delete to remove the file which is called by Eve, it could never be executed, since it is not part of the session type.

3.7. Postconditions

Our session type specification language does permit us to describe communication between active objects and aspects of cooperative scheduling. However, it does not allow us to reason about the effects of method calls on an object's state. In [29], Kamburjan et al. introduced a variation of session types for active objects which allow to specify pre- and postconditions on an object's state for method calls. They also developed a type system for the static verification of these conditions.

Due to the time constraints this thesis is subject to, we do not implement static verification of such conditions. Nevertheless, since our dynamic enforcement methods already modify the AST of ABS models, we extend our session type language with postconditions for calls and verify them at runtime.

Within our Evaluation chapter, we give an example application of session types with postconditions, see Section 5.2.

3.7.1. Modified Session Type Syntax

We modify the syntax of global session types so that all interactions can optionally carry a postcondition in double square brackets “⟦·⟧”. The remaining syntax stays the same:

$$\begin{array}{rcl} \boxed{\mathrm{G}} & ::= & 0 \xrightarrow{f} p\colon m\,[\llbracket e_p \rrbracket] \;\mid\; \boxed{\mathrm{G}}.\boxed{\mathrm{g}} \\ \boxed{\mathrm{g}} & ::= & p \xrightarrow{f} q\colon m\,[\llbracket e_p \rrbracket] \;\mid\; \ldots \end{array}$$

This postcondition must be an ABS pure expression e_p, see [27] and [11], though we do not allow all pure expressions supported by Full ABS. See Appendix G.1 for a full list of supported expressions. For object local types, the receiving type has also been extended with an optional postcondition:

$$\begin{array}{rcl} \boxed{\mathrm{L}} & ::= & \boxed{\mathrm{l}}[.\boxed{\mathrm{L}}] \\ \boxed{\mathrm{l}} & ::= & 0?_f m\,[\llbracket e_p \rrbracket] \;\mid\; p?_f m\,[\llbracket e_p \rrbracket] \ldots \end{array}$$

Since we only need object local types for applying the postconditions to an ABS model, we do not extend the syntax of method local types.

3.7.2. Changes to CST Validation, Projection and Automaton Generation

The annotation of global types with pre- and poststates during CST Validation must now preserve postconditions for all types and otherwise ignore them. Object local projection is slightly adapted to preserve postconditions when projecting an interaction or initialization to a receiving type:

$$\mathit{project}^q\left(p \xrightarrow{f} q\colon m\,\llbracket e_p \rrbracket\,\left\langle \sigma_{\mathit{pre}}, \sigma_{\mathit{post}} \right\rangle\right) = p?_f m\,\llbracket e_p \rrbracket\,\left\langle \sigma_{\mathit{pre}}, \sigma_{\mathit{post}} \right\rangle .$$

Method local projection removes all annotated postconditions.

The generation of automata is altered so that *InvocREv* transitions are annotated with the postcondition of the receiving type they have been generated from, if present:

$$\mathit{genAutomaton}(p?_f m\,\llbracket e_p \rrbracket, \ldots) = ((\{q_0, q_2\}, q_0, \{(q_0, (\mathit{InvocREv}(m), r)\llbracket e_p \rrbracket, q_1)\}, \{q_1\}), \ldots$$

Postcondition annotations are preserved by all other transformations of *genAutomaton* and εNFAToDFA. The latter one treats postconditions as part of the transition label when grouping transitions.

3.7.3. AST Extensions

Now that postconditions are carried by the *InvocREv* transitions of our Session Automata, we can just extend the method modification steps of Section 3.5.6 to check them at runtime:

Let $A = (Q, q_0, \Delta, F)$ be the automaton being integrated into a class. Let m be the method currently being modified. Then

$$T = \{(q_1, (\mathit{InvocREv}(m), r)[\![e_p]\!], q_2) \in \Delta\}$$

is the set of *InvocREv* transitions of m with a postcondition. We now add three steps after step 3 to the modifications listed in the subsection **_InvocREv_ Transitions** of Section 3.5.6:

4. a new local variable `invocState` is initialized at the beginning of m before the `case`-statement we inserted. It remembers the automaton state the method has been activated in:

   ```
   Int invocState = this.q;
   ```

5. If the method contains a return statement `return e;` at its end, we initialize a new variable `result` with e in front of the return statement. Then we replace the expression of the return statement with a use of the new variable:

   ```
     ...
     return e;
   }
   ```

   $\Longrightarrow$

   ```
     ...
     Int result = e;
     return result;
   }
   ```

   The reason for this modification is that our postcondition checks shall be the last actions performed in a method. Therefore, we need to move e out of the `return`-statement since its evaluation may have side-effects.

6. We insert the following `case`-statement at the end of m, or right before its `return`-statement, if there is one:

```
case invocState {
   $(\mathit{toInt}(q')$ => $\langle \mathit{conditions}\ q' \rangle$;$)_{(q',v,q'') \in T}$
   _ => skip;
}
```

Thanks to the `invocState` variable, the **case**-statement can determine through which *InvocREv* transition the current execution of m has been originally activated and thus derive which postcondition applies. The branches $\langle \mathit{conditions}\ q' \rangle$ each contain an assertion checking the postcondition of this transition. It should be noted, that there can be more than one *InvocREv* transition for m in q' if automaton A is not data-deterministic. In this case, a warning is already printed by step 3 and we simply check the postconditions of all possible *InvocREv* transitions:

```
$\langle \mathit{conditions}\ q' \rangle =$ {
   $($assert $e_p$;$)_{(q',(\mathit{InvocREv}(m),r)[\![e_p]\!],q'') \in T}$
}
```

As an illustration, we give the result of applying these modifications to method `start` of the example model of Section 5.2 in Listing 11 of Appendix F.

4. Implementation

In this chapter we give a rough overview of the structure of our application and software tools we utilized. The Concept chapter 3 already explains in great detail how the individual modules of our application (CST Validation, Projection, Static Verification, Dynamic Enforcement) operate. Moreover, we were able to directly transfer our formalizations of the Concept chapter, e. g. the $execute_{\mathbb{C}}$, *project*, *genAutomaton* functions etc., into an implementation without the need of an imperative translation or object-oriented layer. This is because we largely make use of the functional programming paradigm in our implementation. Therefore, we do not discuss the specifics of the implemented functions here.

In Section 4.1, we first credit some tools and libraries we used for the implementation. Next, the workflow of our implementation is presented in Section 4.2. Section 4.3 introduces our parser for session types and their representation within our application. Section 4.4 gives some insight into how session type validation has been implemented using our Configurable Session Type Validation method. In Section 4.5 we describe the realization of the type system used for static verification. How AST modifications are performed to achieve dynamic enforcement of scheduling policies and runtime checks of postconditions is already explained thoroughly in Sections 3.5.6 and 3.7.3 so we do not discuss this here.

4.1. Tools and Libraries

The SDS-tool has been written in the programming language Kotlin [26] since we required a language which is interoperable with the Java based *abstools* compiler but also encourages a functional programming style. This way, we could implement the components of our tool so that they closely resemble our formalization.

Our tool is not an extension of the *abstools* compiler but we use it as a library instead. Thus, our tool can be maintained separately from the ABS compiler. However, as explained in Section 3.5.5, we still had to modify it slightly[1].

We parse session types using the parser generator ANTLR [40], see also Section 4.3. Command line arguments are handled using the *picocli* library [41]. All other used utility libraries are credited in Appendix G.2.

4.2. Workflow

Figure 3.3 of Section 3.1 depicts all important components of our tool as well as their interactions on a high level. In this section, we briefly elaborate on this workflow on a slightly more technical level. When a user calls our tool on the command line, they need to provide an ABS model and a set of global session types as source files. It then outputs a compiled Erlang version of the model with schedulers and postcondition checks. More specifically, our main class executes the steps listed in Algorithm 1, see below.

We leave the parsing and data type checking of ABS models to the *abstools* compiler, see Lines 3 and 4. Lines 5 to 8 realize the preprocessing of session types and their static verification for an ABS model. Line 9 realizes our dynamic enforcement techniques. Since we modify the AST, we again check it using *abstools* in Line 10. This is partly to confirm that there are no type errors in our modifications but also necessary because *abstools* must rewrite some parts of an AST during typechecking to be able to compile it. If any step fails (data type checking, static verification, ...), then the tool aborts execution.

[1] You can find the modified branch of *abstools* at [43].

Algorithm 1 Main compile method of the SDS-tool (`Compile.kt`).

```
procedure compile(cli arguments (*.abs and *.st files))
    parse command line arguments with Pico CLI.
    parse *.abs files using abstools.
    typecheck and rewrite resulting AST using abstools[2].
    buildTypes(*.st)                                    ▷ preprocessing of session types
    for each global session type do                     ▷ static verification
        apply type system rule Model to model AST
        apply additional checks of Section 3.4.4
    apply modifications of Section 3.5.2 and Section 3.7.3 to AST.
    typecheck and rewrite modified AST using abstools
    compile modified AST to Erlang with abstools

procedure buildTypes(*.st files)
    parse *.st files using ANTLR
    apply $execute_{\mathbb{C}}$, the Configurable Session Type Validation
    check no actor participates in more than one global type
    perform object projection using project

    return analyzed global and local types
```

4.3. Session Type Parser

Users should be able to provide session type specifications to our tool which do not require special symbols like the ones we introduced in Section 2.3. Thus, we developed an ASCII based version of the grammar of Sections 2.3.1 and 3.7 which we parse using the ANTLR tool [40]. The following table gives a short overview on how each kind of global session type is represented in this new syntax:

Type	Representation
$0 \xrightarrow{\mathtt{f}} \mathtt{P}\colon \mathtt{m}[\![\mathtt{pure\text{-}exp}]\!]$	`0 -f-> P:m<pure-exp>`
$\mathtt{P} \xrightarrow{\mathtt{f}} \mathtt{Q}\colon \mathtt{m}[\![\mathtt{pure\text{-}exp}]\!]$	`P -f-> Q:m<pure-exp>`
$\mathtt{P} \downarrow \mathtt{f}\,(\mathtt{C})$	`P resolves f with C`
$\mathtt{P} \uparrow \mathtt{f}\,(\mathtt{C})$	`P fetches f as C`
$\mathit{Rel}(\mathtt{P}, \mathtt{f})$	`Rel(P, f)`
Skip	`skip`
$\mathtt{P}\,\{\ldots\}$	`P {...}`
$(\ldots)^*$	`(...)*`
$G_1.G_2$	`G1.G2`

The allowed characters for actors `P` and `Q` follow the grammar for class identifiers of ABS. The same applies to futures `f`, methods `m` and data constructors `C`. For full examples of this user input syntax, see Chapter 5.

Internally, we represent session types after the parsing process as a recursive algebraic data type (ADT). More precisely, we use the Kotlin representation of this concept, which is a *sealed data class* with a subclass for each kind of session type. For example, this is the class hierarchy implementing global session types:

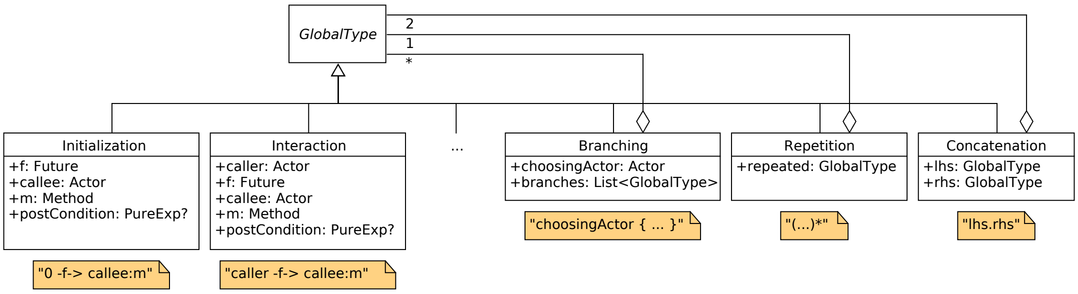


Figure 4.1.: Class hierarchy implementing global types.

4.4. Configurable Session Type Validation

The concept of a configurable session type analysis structure of Section 3.2 has been implemented as an interface containing all methods a configurable analysis must support, see Figure 4.2. Consequently, each of our analyses has been realized as a class implementing this interface. A analysis's state is an instance of the class and the initial state σ_0 defined by its default constructor. The Combined Analysis is a class composed of the other ones and delegates all calls to them. The validation process therefore is modular and can be extended by adding another class to the Combined Analysis.

In our concept, the functions and predicates of an analysis are undefined if a session type is invalid. In our implementation, an analysis's methods instead throw an exception in this case. Our tool then displays a description of the issue to the user and aborts. The $\mathit{execute}_{\mathbb{D}}$ function is implemented just like the formalization. It returns another session type ADT containing instances of CombinedAnalysis as pre- and poststates.

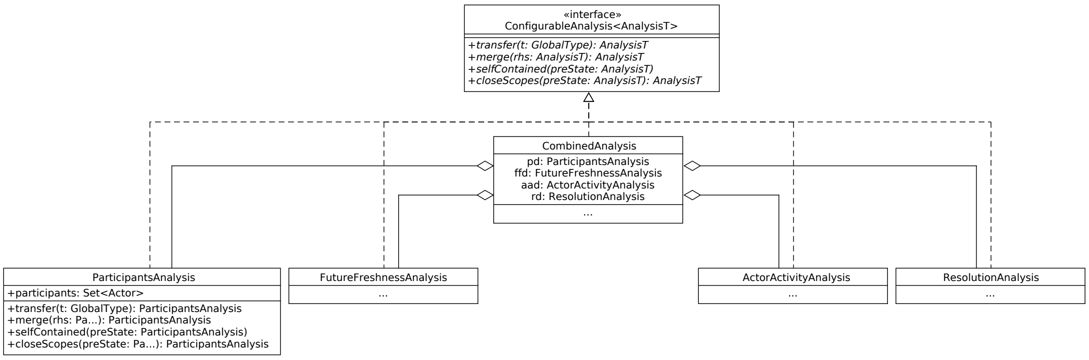

«interface»
ConfigurableAnalysis<AnalysisT>
+transfer(t: GlobalType): AnalysisT
+merge(rhs: AnalysisT): AnalysisT
+selfContained(preState: AnalysisT)
+closeScopes(preState: AnalysisT): AnalysisT
CombinedAnalysis
pd: ParticipantsAnalysis
ffd: FutureFreshnessAnalysis
aad: ActorActivityAnalysis
rd: ResolutionAnalysis
...
ParticipantsAnalysis
+participants: Set<Actor>
+transfer(t: GlobalType): ParticipantsAnalysis
+merge(rhs: Pa...): ParticipantsAnalysis
+selfContained(preState: ParticipantsAnalysis)
+closeScopes(preState: Pa...): ParticipantsAnalysis
FutureFreshnessAnalysis
...
ActorActivityAnalysis
...
ResolutionAnalysis
...

Figure 4.2.: Implementation of configurable session type analyses.

4.5. Static Verification

We implemented the structural rules Model, Classes, Class, Method and Main of our type system (see Section 3.4.3) as a set of functions which apply all checks described by them.

However, this approach is not applicable for the statement rules. Whereas for structural units it is immediately clear by the type of the AST node (Model, Class, Method, Main Block) which rule must be fulfilled, this is not the case for statements. Instead, oftentimes all conditions (except for recursive rule application) of a set of rule candidates must be evaluated to decide which rule must apply.

Therefore, we split up the representation of a statement rule into a guard method and an invoke method. The first one checks its conditions without rule recursion and the second one applies the recursion. Statement rules have thus been realized as singletons implementing this common interface:

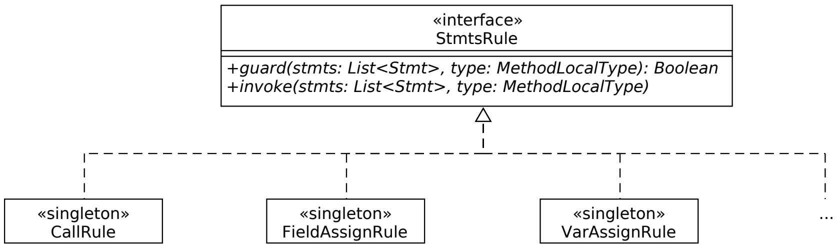

«interface»
StmtsRule
+guard(stmts: List<Stmt>, type: MethodLocalType): Boolean
+invoke(stmts: List<Stmt>, type: MethodLocalType)
«singleton»
CallRule
«singleton»
FieldAssignRule
«singleton»
VarAssignRule
...

Figure 4.3.: Type system rule interface and rule singletons.

Furthermore, a function checkStmts has been implemented to conduct verification of statements using these rule singletons. It tests the guard method of all rule singletons for the statements to be verified and their session type. Then the invoke method of the singleton whose guard returns true is applied[3]. A model does not fulfill its session type if no rule is applicable. This will abort execution of the SDS-tool with an error. The invoke method in turn calls checkStmts again if the rule is not axiomatic. Otherwise recursion stops:

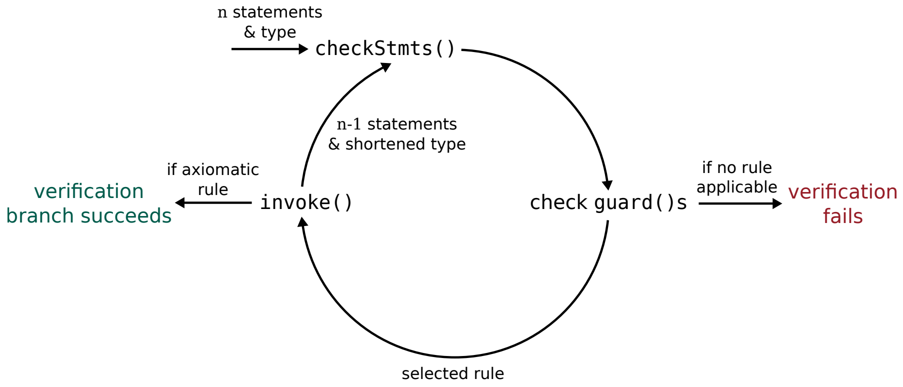


Figure 4.4.: Workflow of method body verification.

[3] We adapted the rules for the implementation slightly so that at most one of them can apply to a statement.

5. Evaluation

In the first part of this chapter we apply our tool to examples of distributed systems from [30] and [29]. We demonstrate that our tool solves the problems posed by the examples. We also point out some restrictions of the tool which become apparent when implementing these examples. Furthermore, an implementation of Example 1 of Section 2.3 is given to showcase the branching and repetition features of session types not covered in the other two examples.

In the second part of this chapter, we evaluate how our dynamic verification techniques, namely schedulers, affect the execution performance of ABS models.

For information on how these experiments have been conducted and instructions to reproduce the results of this chapter, see Appendix B.

5.1. Example 1: Ordered Activations

In [30], the need for schedulers as a means to enforce an execution order is illustrated with an example similar to the following one:

A grading system consists of a computation server C, a report generator R, a service desk D and a student S. The computation server's task is to compute the student's grade, send it to the report generator and notify the student about the new grade. The report generator renders a human-readable report from the server's data which can take some time if the generator is busy. The generator then publishes the report at the service desk. After receiving the notification from the computation server, the student will request their grade from the service desk. However, the desk may handle this request only after it has processed the publication of the report. Therefore, if the student's request arrives before the report, the service desk's scheduler must not activate the request until it has processed the message from the report generator. The system and its messages have been illustrated in Figure 5.1.

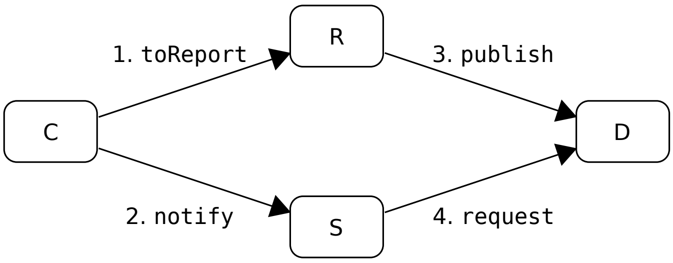


Figure 5.1.: Visualization of the grading system and the messages between its active objects. Message 3 must be processed by D before message 4. The illustration has been adapted from Figure 1 of [30].

The protocol described above is captured by the following session type. Its representation in the input format accepted by our tool is displayed on the right:

$$
\begin{array}{l}
0 \xrightarrow{f} C\colon \texttt{compute}.\\
\quad C \xrightarrow{f_{\mathit{toReport}}} R\colon \texttt{toReport}.\\
\quad C \xrightarrow{f_{\mathit{notify}}} S\colon \texttt{notify}.\\
\qquad R \xrightarrow{f_{\mathit{publish}}} D\colon \texttt{publish}.\\
\qquad R \downarrow f_{\mathit{toReport}}.\\
\qquad\quad D \downarrow f_{\mathit{publish}}.\\
\qquad S \xrightarrow{f_{\mathit{request}}} D\colon \texttt{request}.\\
\qquad\quad D \downarrow f_{\mathit{request}}.\\
\qquad S \uparrow f_{\mathit{request}}.\\
\qquad S \downarrow f_{\mathit{notify}}.\\
C \downarrow f
\end{array}
$$

```
0 -f-> ComputationServer:compute.
  ComputationServer
    -fToReport-> ReportGenerator:toReport.
  ComputationServer
    -fNotify-> Student:notify.
      ReportGenerator
            -fPublish-> ServiceDesk:publish.
      ReportGenerator resolves fToReport.
        ServiceDesk resolves fPublish.
  Student -fRequest-> ServiceDesk:request.
    ServiceDesk resolves fRequest.
  Student fetches fRequest.
  Student resolves fNotify.
ComputationServer resolves f
```

Next, we present an ABS model of the grading system (we leave out the interfaces for brevity):

```
class ComputationServer
 (ReportGeneratorI r, StudentI s)
 implements ComputationServerI {
❶ Fut<Unit> fToReport;
  Fut<Unit> fNotify;

  Unit compute() {
    Int grade = random(6) + 1;

❷   this.fToReport =
      r!toReport(grade);
❸   this.fNotify = s!notify();
  }
}

class Student (ServiceDeskI d)
    implements StudentI {
  Fut<String> fRequest;

  Unit notify() {
❹   this.fRequest = d!request();
❺   String gradeReport =
      this.fRequest.get;
  }
}

class ReportGenerator
    (ServiceDeskI d)
    implements ReportGeneratorI {
  Fut<Unit> fPublish;

  Unit toReport(Int grade) {
❻   duration(1, 1);
    this.fPublish = d!publish(
     ”New␣grade:␣”+toString(grade)
    );
  }
}
```

```
class ServiceDesk
    implements ServiceDeskI {
  Maybe<String> report = Nothing;

  Unit publish(String report) {
❼   println(”publish”);
    this.report = Just(report);
  }

  String request() {
    println(”request”);
    return fromJust(this.report);
  }
}

{
❽ ServiceDeskI d =
    new ServiceDesk();
  ReportGeneratorI r =
    new ReportGenerator(d);
  StudentI s = new Student(d);
  ComputationServerI c =
    new ComputationServer(r, s);

  await c!compute();
}
```

Actors are implemented as classes of the same name (see Section 3.4.2). Calls and **get**-expressions realize the interacting ($p \xrightarrow{f} q\colon m$) and fetching ($p \uparrow f$) components of the type.

This model can be checked against the session type by our SDS-tool and be compiled to Erlang. As intended, our tool reports an error during static verification if the model does not fulfill the session type. For example, if statements ❷ and ❸ or ❹ and ❺ were to be swapped or removed, the user is informed and the compilation aborts. The **`duration`**-statement in ❻ of *Real-Time ABS* [5] allows us to simulate a slight delay during generation of a report. Actions like ❼ which have no effect on the session are permitted at any point.

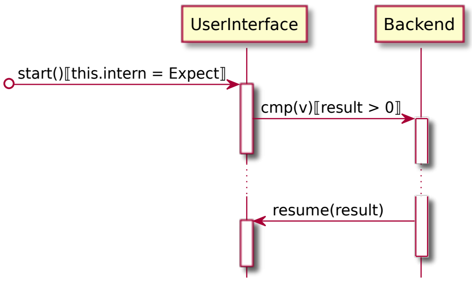
This example also demonstrates that our tool imposes some restrictions on modeling. For instance, futures of the session have to be stored in fields as in ❶. Moreover, ❽ shows that all actors have to be instantiated in the main block and references be shared as initialization parameters. This is a more severe restriction since it implies that at the beginning of a session not all actors can have a reference to all other actors, e. g. the service desk initially can not have a reference to the student. Nevertheless, if required, actors can still share a reference to themselves as parameters of calls. However, those calls have to be added to the session type and the model.

The schedulers our tool adds to the model during compilation solve the invocation ordering issue of the grading system described by [30]. Out of 100 executions, `request` is never activated before `publish`[1]. On the other hand, if we deactivate the dynamic enforcement of our tool, in 100 out of 100 executions the ordering is violated.

5.2. Example 2: Heap Communication

We use an example from [29] to demonstrate the runtime postcondition checking feature of our tool. Imagine a user interface U which executes costly computations on a backend server B to stay responsive. After the method `start` is called on U, U asks B to perform the costly computation `cmp`. Before completing the method it then sets a flag in its heap which marks that it expects a callback from B with the result. Therefore, this is an example of communication between methods of an object via the object's heap. When B finishes the computation, which must always have a positive result, it calls `resume` on U with the value of the result as parameter. It must be ensured that at all times when `resume` is invoked the aforementioned flag is set. Figure 5.2 gives an overview of this protocol.

[1] We inspect the output of the model and check whether "request" is ever printed before "publish".



$$
\begin{array}{l}
0 \xrightarrow{f_{start}} U\colon \text{start}\ \llbracket \textbf{this}.\text{intern} \ == \text{Expect} \rrbracket\ . \\
\quad U \xrightarrow{f_{cmp}} B\colon \text{cmp}\ \llbracket \text{result} \ > 0 \rrbracket\ . \\
U \downarrow f_{start} . \\
\quad B \xrightarrow{f_{resume}} U\colon \text{resume}. \\
\quad B \downarrow f_{cmp} . \\
\qquad U \downarrow f_{resume}
\end{array}
$$

Figure 5.2.: Sequence diagram of the interactions between user interface and backend on the left. This illustration has been adapted from [29]. On the right, a session type formalization of the scenario is depicted.

We give the input format of the session type for our tool below:

```
0 -fStart-> UI:start<this.intern == Expect>.
  UI -fCmp-> Backend:cmp<result > 0>.
UI resolves fStart.
  Backend -fResume-> UI:resume.
  Backend resolves fCmp.
    UI resolves fResume
```

We can see that postconditions can be used to reason about the state of fields (`this.intern`) as well as local variables (`result`). Next, we present the ABS model implementing the scenario:

```
data TState = Init | Expect;

class UI (BackendI b)
    implements UII {
❶ TState intern = Init;
  Fut<Unit> fCmp;

  Int resume(Int result) {
    // ...
    return if (intern != Expect)
      then -1 else 1;
  }

  Unit start(Int v) {
    this.fCmp = b!cmp(this, v);
❷   intern = Expect;
  }
}
```

```
class Backend
    implements BackendI {
  Fut<Int> fResume;

  Unit cmp(UII u, Int v) {
❸   Int result = 42*v;
    this.fResume
      = u!resume(result);
  }
}

{
  BackendI b = new Backend();
  UII u = new UI(b);

❹ await u!start(42);
}
```

The field **`this`**`.intern` at ❶ models the flag which is set in ❷ to signify that a callback from B is to be expected. Our tool ensures that the execution of `u` and `b` respectively aborts immediately when reaching the end of `start` or `cmp` if the postconditions encoded in the type were to be violated. That is the case if for example statement ❷ were to be removed or the value `42` in ❹ were to be replaced with a negative value.

Again, we can also observe some limiting aspects of our tool. For example, postconditions can not be used to check the value of the local variable `result` before passing it to `resume`. Also, since we support no preconditions, such a check can not be placed at the beginning of `resume`. Therefore, `resume` can execute with an invalid value and only the `b` object will be affected by the failing postcondition.

Furthermore, the usual limitations of runtime verification apply, that is, a static verification of postconditions could detect that `cmp`'s postcondition can not be proven to hold for all inputs. Whereas our dynamic checks would only report this issue if the model is executed with negative inputs.

Lastly, we want to point out another modeling restriction. In the original example of [29] there is an interface server I placed between the UI and the backend. Instead of passing on the result computed by the backend to U, I hands over the future for that computation. Since our tool only allows interactions with futures stored in special fields, see Section 3.4.3, using futures which have been communicated between objects is not supported.

5.3. Example 3: Notification Service

We now give an implementation of Example 1 of Section 2.3.1 to demonstrate repeating and branching types. For information on the implemented scenario and a session type formalization, please see Section 2.3.1.

The session type is encoded into the input syntax of our tool like this:

```
0 -fInit-> NotificationService:init.
(
  NotificationService -fCheckMail-> MailServer:checkMail.
  Rel(NotificationService, fCheckMail).
  MailServer {
    MailServer resolves fCheckMail with NewMail.
      NotificationService fetches fCheckMail as NewMail.
      NotificationService -fPopup-> UI:popup.
        UI resolves fPopup,
    MailServer resolves fCheckMail with NoMail.
      NotificationService fetches fCheckMail as NoMail
  }
)*.
NotificationService resolves fInit
```

We implemented the scenario as the following ABS model:

```
data MailMsg =
    NewMail(String mail) | NoMail;

class NotificationService
  (MailServerI m, UII u)
  implements NotificationServiceI
{
  Fut<MailMsg> fCheckMail;
  Fut<Unit> fPopup;

  Unit init() {
    Int i = 0;
❶   while (i < 10) {
      this.fCheckMail =
        m!checkMail();
      await this.fCheckMail?;

      MailMsg response =
        this.fCheckMail.get;
❷     case response {
        NewMail(mail) =>
          this.fPopup =
            u!popup(mail);
        NoMail => skip;
      }
      i = i + 1;
    }
  }
}
```

```
class MailServer
    implements MailServerI {
  MailMsg checkMail() {
    MailMsg result = NoMail;
❸   if (random(2) == 1) {
      result =
        NewMail(”Hello␣World!”);
    }
    return result;
  }
}

class UI implements UII {
  Unit popup(String mail) {
    println(”You␣got␣mail:␣” +
      mail);
  }
}

{
  MailServerI m =
    new MailServer();
  UII u = new UI();
  NotificationServiceI n =
    new NotificationService(m, u);

  await n!init();
}
```

The repetition `(...)*` within the type has been implemented as the **`while`**-loop at ❶. For this example, we choose to restrict the execution to 10 iterations so that the model terminates. An infinite loop would also have been accepted by our static verification.

In the branching type `MailServer {...}`, actor `MailServer` does decide on which branch the session progresses and `NotificationService` must follow this decision. This is achieved by reading the result of the call to `MailServer` and forking the control flow by applying the **`case`**-statement at ❷ to the value. Please note, that our type system only accepts **`case`**-statements for this purpose, therefore, **`if`**-statements could not have been used here, see also Section 3.4.3.

On the other hand, for the deciding actor, we allow much more modeling freedom. As long as all possible control flows of a deciding method are typed by one branch of the session type, it is accepted. In this model, we randomly decide whether there should be mail or not, see ❸. It should be noted that our static verification would accept the model if a value created with a third constructor, not `NewMail` or `NoMail` were to be returned but it would cause an exception at ❷ at runtime. See also our remarks regarding the type system rules Offer and ReturnMsg in Section 3.4.3.

5.4. Performance Evaluation

To assess the performance impact of our schedulers, we designed a minimal example of communicating active objects and extend it in different ways to observe the impact on our performance metrics. We also give a brief overview on how the performance of the slightly more complex model of Section 5.3 is affected by schedulers.

We start out with an ABS model containing two actors, P and Q. P is repeatedly calling two methods on Q which have to be activated in the right order:

$$0 \xrightarrow{f} P\colon \mathtt{m}.\left(P \xrightarrow{f_{m1}} Q\colon \mathtt{m1}.Q \downarrow f_{m1}.P \xrightarrow{f_{m2}} Q\colon \mathtt{m2}.Q \downarrow f_{m2}\right)^{*}.P \downarrow f$$

We list the source code for this type which is read by our tool in Listing 12. This is the ABS model implementing the type:

```
module Model;

interface PI {
  Unit m();
}

interface QI {
  Unit m1(Int i);
  Unit m2(Int i);
}

class P (QI qRef) implements PI {
  Fut<Unit> fm1;
  Fut<Unit> fm2;

  Unit m() {
    Int i = 0;
    while (i < ⟨times⟩) {
      this.fm1 = qRef!m1(i);
      this.fm2 = qRef!m2(i);

      i = i + 1;
    }
  }
}
```

```
class Q implements QI {
  Unit m1(Int i) {
    println("m1(" +
    toString(i) + ")");
  }

  Unit m2(Int i) {
    println("m2(" +
    toString(i) + ")");
  }
}

{
  QI q = new Q();
  PI p = new P(q);

  await p!m();
}
```

We call the model the “plain model” when being compiled without adding schedulers and otherwise the “enforcement model”. The following performance metrics are measured by us for the execution of the plain and enforcement models for a varying number of repetitions:

Execution Time This is the required time in user mode for the full execution of the model.

Max. Memory RSS the maximum memory resident set size (RSS) of the Erlang process executing the model. The resident set size is the amount of main memory occupied by a process. Therefore, it does not include used swap space etc.

Order Violations To set the performance penalties of schedulers into perspective, we also need a metric of how severely the plain model violates the order of invocations without a scheduler.

In this simple model, invocations of `m1` and `m2` should be in the following order when being executed sequentially, that is

$$\texttt{m1(0)m2(0)m1(1)m2(1)}\ldots\texttt{m1(}\langle \mathit{times}\rangle - 1\texttt{)m2(}\langle \mathit{times}\rangle - 1\texttt{).}$$

However, the session type does not require this exact order but only that invocations of `m1` and `m2` alternate, since it does not specify parameter values.We recorded the model's output and compared it against the expected alternating and sequential orders using the Levenshtein edit distance [35]. This gives us an approximate measure of how severely the intended order is violated.

Scheduling Delays We record how often the scheduler of `Q` hinders the progress of the model's execution. That is, we measure how often the scheduler could not select an activation compared to the overall number of times the scheduler has been called[2].

For all measured values, we computed the average over 10 executions of the model to mitigate the influence of background processes and the operating system on our experiments.

5.4.1. Phase I: Varying Repetitions

During the first line of experiments, we exclusively adjust the **`while`**-condition ⟨*times*⟩ to increase the number of repeated calls of `m1`and `m2`

Execution Time Figure 5.3 shows that the execution time is stable at around 0.22 seconds for both the plain and the enforced model until around 100 repetitions. Our dynamic enforcement techniques also seem to only slightly increase execution time in this range, there are even cases, where the execution time decreased. Starting from the datapoint of 300 repetitions, execution time clearly increases and it grows more rapidly for the enforcement model. At the datapoint of 500 repetitions, the enforcement model already takes more than twice as much time to execute.

[2] We modified the generation of scheduling functions so that they report status messages which we collected during the model's execution.

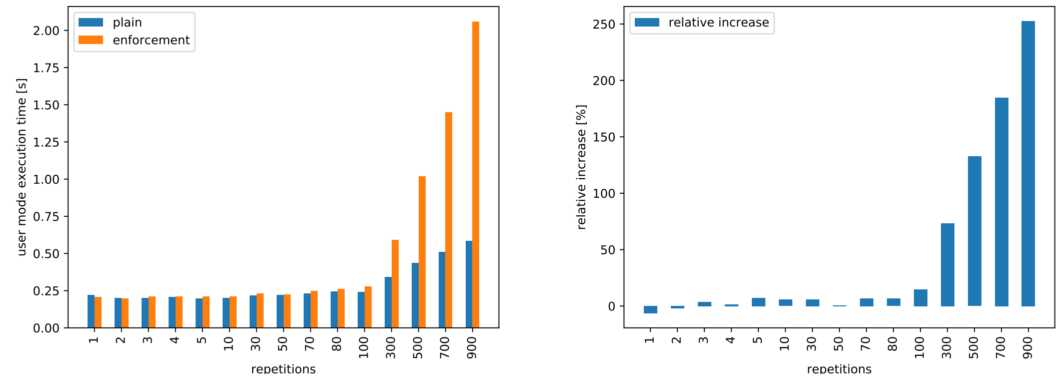


Figure 5.3.: User-mode execution time. Left plot displays absolute values, right one relative increase in time when using dynamic enforcement. Source data see Table C.1.

Max. Memory RSS Regarding memory consumption, similar observations as with the execution time can be made, see Figure 5.4. However, it seems at least for this simple model, schedulers do not affect memory consumption as drastically as execution time, since we observe “only” an increase of about 22% at 900 repetitions.

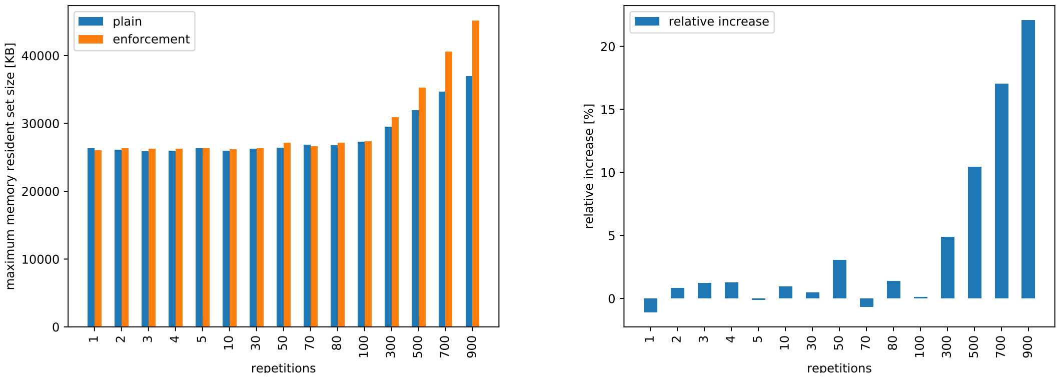


Figure 5.4.: Maximum memory set size. Left plot displays absolute values, right one relative increase in time when using dynamic enforcement. Source data see Table C.2.

Order Violations We observe that up to 5 repetitions almost no deviations from the expected invocation sequence occur and starting from 30 iterations they become increasingly common. For the enforcement model, no deviations from the expected order are observed (see Table C.3). This supports Theorem 3.5.1. There have also no deviations been measured when comparing the invocations of the enforcement model against the strictly sequential order. We presume the Erlang backend only slightly shuffles the delivery of consecutive calls and enforcement of the alternating order of methods is sufficient to restore the sequential order.

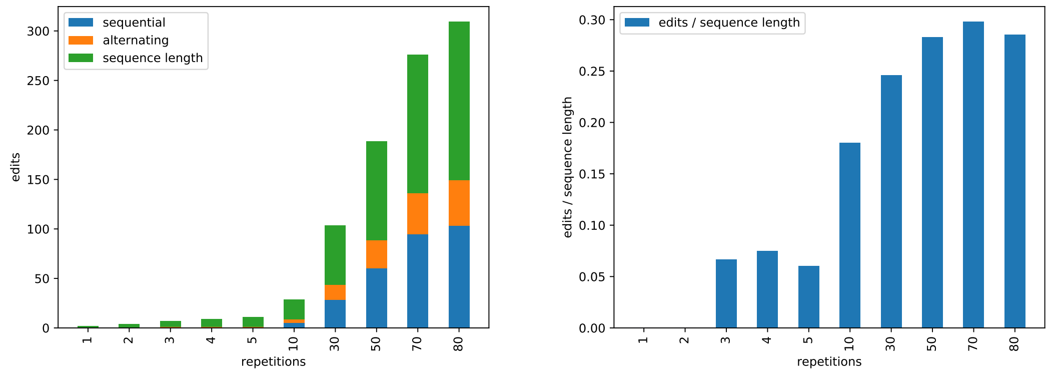


Figure 5.5.: Levenshtein distance of observed and expected invocation sequence. The absolute number of edits when comparing against the sequential or alternating invocation order is displayed left for the plain model. The relative number of edits per sequence length is plotted on the right side. Source data see Table C.3.

Scheduling Delays For this model, the scheduler of `Q` can always select an activation, see Figure 5.6 and Table C.4.

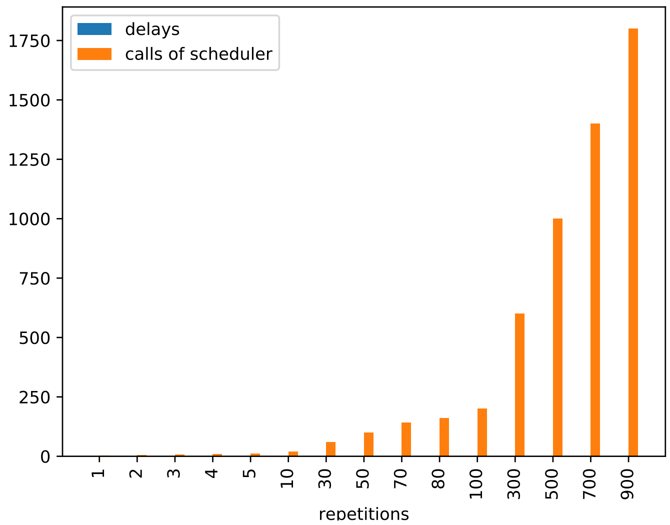


Figure 5.6.: Calls of the scheduler of `Q`. A "delay" is counted for every time the scheduler is called and it can not select an activation. Table C.4.

The following questions remain from the above observations:

- Await statements allow to enforce an invocation order without using schedulers but are more restrictive since they synchronize callers with callees, stalling progress in both objects. Schedulers on the other hand would only potentially introduce delays in the callee and do not require the calling computation to be suspended.

  Therefore, the question arises, how the execution time compares between the plain model and the enforcement model when the order of invocations is guaranteed by **`await`**-statements.

  We explore this question in Section 5.4.2.

- We measured no scheduling delays, likely because the calls immediately arrive at the target object. In a more complex model, there might be some delay between calls because the caller is performing other tasks and computations between them. This in turn should cause delays in a callee object whose scheduler has to wait for the next correctly ordered activation to be available. Meanwhile, the plain model can progress by activating just any method. This hypothesis is tested in Section 5.4.3.

5.4.2. Phase II: Await Statements

For this experiment, we insert **await**-statements after every call to Q:

```
    ...
    this.fm1 = qRef!m1(i);
    await this.fm1?;
    this.fm2 = qRef!m2(i);
    await this.fm2?;
    ...
```

Listing 10: Part of the model where we insert additional **await**-statements.

The session type is also adapted accordingly by adding $Rel(P, \ldots)$ types.

As we can see in Figure 5.7, there is now little difference in execution time between the plain model and the enforcement model. The overhead amounts to at most 13% in our experiments and it does not grow as rapidly for a higher number of repetitions. Also the measured execution times for the enforcement model are often lower than the ones measured in Phase I, see Figure 5.3. Thus, we presume, that the application of our schedulers does not produce much execution time overhead when using **await**-statements to enforce an ordering, at least in a simple example like this one. However, it does seem synchronizing executions using **await**-statements requires less execution time than solely relying on schedulers.

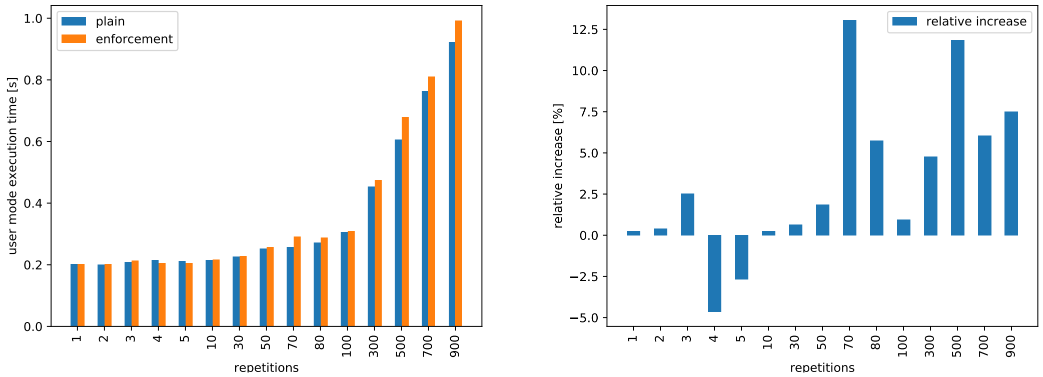


Figure 5.7.: User-mode execution time when using `await` statements. Left plot displays absolute values, right one relative increase in time when using Dynamic Enforcement. Source data see Table C.5.

5.4.3. Phase III: Delayed and Unordered Calls

To simulate calls arriving out of order and with delays at P, we insert `duration`-statements (see [5]) after every call and reverse the order of calls in the ABS model. This requires us to disable static verification for this experiment:

```
...
this.fm2 = qRef!m2(i);
duration(1, 1);
this.fm1 = qRef!m1(i);
duration(1, 1);
...
```

As expected, we now observe calls to the scheduler of Q which result in no activation. Their number does not grow as fast as the overall number of calls with increasing repetitions. Furthermore, compared to the plain model, there can no execution time overhead be consistently measured, presumably because the scheduling overhead falls within the now increased idle time of Q.

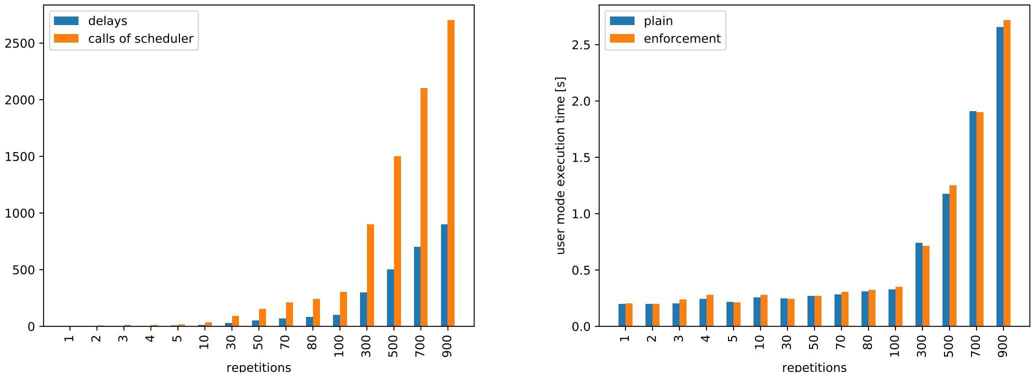


Figure 5.8.: The left plot shows the number of times (“delays”) no activation could be selected by the scheduler of Q compared to the total number of calls. The right plot depicts measured user-mode execution times. Source data see Table C.6.

5.4.4. Performance of More Complex Models

We also measured the execution time and memory metrics for the slightly more complex model of Section 5.3 for an increasing number of repetitions of the **while**-loop, see Figure 5.9.

Overall it shows similar developments in performance metrics as the experiment of Section 5.4.2. This is likely due to the fact, that it also orders invocations with an **await**-statement and has similar communication patterns.

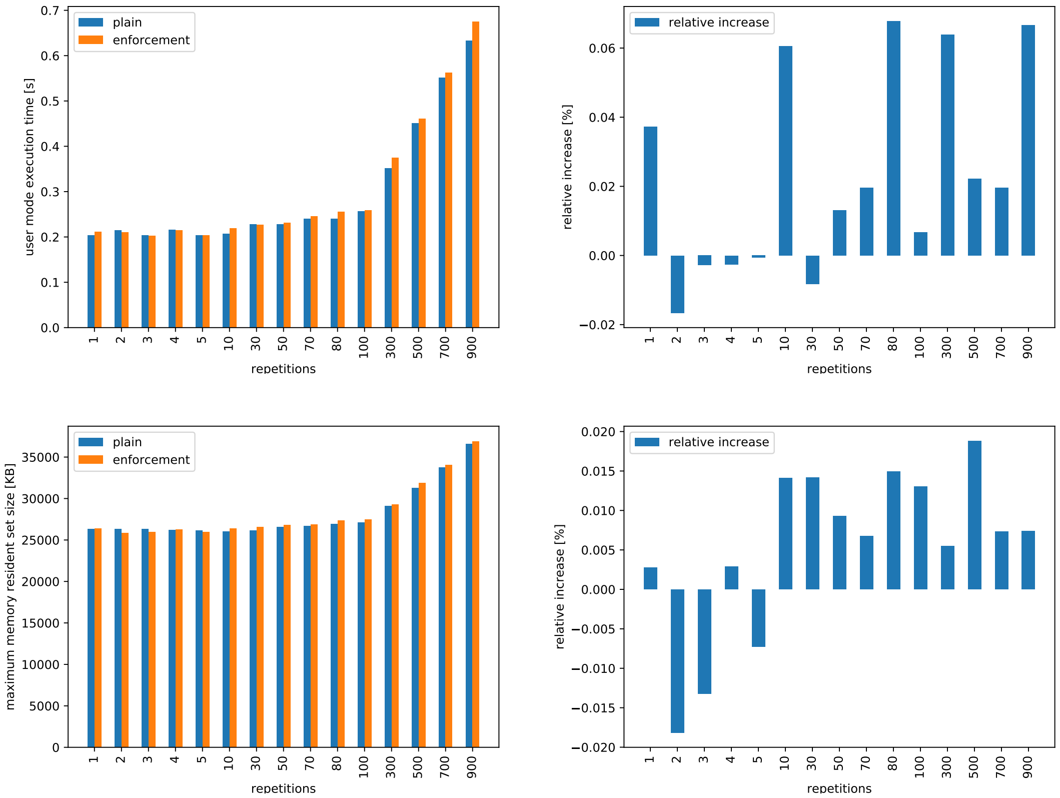


Figure 5.9.: In the upper row: User-mode execution time. In the lower row: Maximum memory set size. The left plots display absolute values, the right ones the relative increase in time/memory when using dynamic enforcement. Source data see Table C.7.

5.4.5. Summary

The following statements only hold in the context of our simple model introduced at the top of Section 5.4, assuming messages arrive without delays.

Our schedulers increase the execution time of models if a high number of calls is issued. The overhead, that is the difference in execution time between the plain and the enforcement model, is stable until a certain number of calls (about $100 \cdot 2 = 200$ calls). Then the overhead grows to increasing multiples of the execution time of the plain model. However, similar increases in execution time are measured if **await**-statements are used to enforce order instead of schedulers. When using **await**-statements, the overhead produced by schedulers is not as noticeable. Also, in absolute values, using **await**-statements for ordering seems more efficient than solely relying on schedulers in regards of execution time.

The maximum memory RSS first stays constant like the execution time and then starts to increase with a high number of calls, though not as rapidly.

The ordering of messages is reliably ensured by our schedulers.

Threats to Validity

The reader should be aware that the conclusions drawn in the above sections are derived from relatively simple models, though they give us an idea of how more complex systems may behave. For example, we only introduced **await**-statements to a single object. It is unclear, which effect a high number of future reactivation transitions in schedulers has when being distributed among multiple actors. The implementation of a case study on larger models with more complex communication patterns remains as future work.

6. Conclusion

In this thesis, we applied existing theoretical work by Kamburjan et al. [30, 28, 29] to implement a software tool which allows to statically verify that the components of an ABS model comply with a session type specification. Furthermore, it extends the model's abstract syntax tree to dynamically enforce specified behavior in the composite model where it can not be statically guaranteed.

To perform the static verification, we *project* the session type on each participating object and method to derive a specification of their local behavior. This local specification is verified using a type system. We developed our own variation of the projection process and type system based on the existing theoretical work. This way, they are suited for a straightforward implementation and can be applied to the current ABS language version as realized by the *abstools* compiler.

We also introduced a new method of validating session type specifications separately from the projection process which is based on Configurable Software Verification [4]. This separation of concerns improves the maintainability of our project and allows our session type language to be easily extended.

Since the order of activations of methods can in many cases not be statically guaranteed, we automatically extend ABS models to dynamically enforce a specified order using schedulers based on Session Automata. Moreover, our session type language based on [30] has been extended with ideas from [29] to include postconditions for method invocations. This allows the user to reason about an object's heap and local variables after the execution of a method. These postconditions are checked at runtime as part of our modifications to ABS models.

The *abstools* compiler has been modified to allow the compilation of modified models to Erlang so that they can be simulated. The impact of schedulers on the performance of such simulations has been investigated. The results suggest that schedulers can have a considerable impact on the execution time of a model. However, such overhead is also produced by the existing method of ensuring an execution order via **await**-statements.

6.1. Related Work

Other Implementations of Session Types We implemented session types as an extension of the ABS modeling language, a language based on active objects. There are many other implementations of session types for different programming languages and a survey of them can be found in [2]. A list of implementations is also being maintained by Fowler in [17]. Some of these implementations are intended for use in the academic context and others try to incorporate session types into *main-stream* languages like Java or C for practical application.

SePi [18] is a language of the first kind which implements a variation of the π-calculus and aims to be a testing ground for theoretic work on session types. It statically checks communication via channels against session types. SePi's session types differ from ours and resemble more closely their original formulation [46, 21], due to SePi's vastly different concurrency model and usage of channels for communication. Messages are data types which can be refined [3] instead of method invocations. There is no notion of cooperative scheduling, like our $\mathit{Rel}(p, f)$ types.

There are also multiple approaches to integrating session types in an object-oriented setting. Session Java [24] is an extension for the Java language which adopts channel-based session types for TCP sockets. For this language, local endpoints are statically verified against session types. However, the compatibility of these local types is checked dynamically upon the connection of sockets. Mool [9], a minimal object-oriented language, employs method calls instead of data type messages as a communication model. Therefore, its session type language is more similar to ours than the other languages. However, they also do not encode cooperative scheduling. In Mool, classes are annotated with a *usage type* which describes the order and structure of method invocations. The state of an object and its usage type determine which methods can be called and calling a method changes the state according to the type. This concept is similar to our schedulers whose automaton states determine the methods which can be called on an object. However, Mool verifies conformance with usage types statically, whereas we require a dynamic solution via schedulers since message ordering is not guaranteed in ABS.

A “Multiparty Session Actor” framework for applying session types to actor-based concurrency models has been developed by Neykova and Yoshida [38] and implemented by emulating actors in Python. There is also a more direct implementation of the framework in the actor-based language Erlang [16]. It monitors communication at runtime using finite-state machines generated from session types. However, their types do not incorporate cooperative scheduling and the monitors do not delay and reorder messages like our schedulers.

Though we discovered session type implementations for process calculi (SePi), functional (Links [36]), imperative (Session C [39]), object-oriented (Mool, Session Java) and actor-based programming languages (Multiparty Session Actors), we were not able to find an implementation where the communication and concurrency models resemble ABS and the session types describe cooperative scheduling.

The languages mentioned above all either have session types built into the language, are extensions of a language with session types like our tool, or perform all verification dynamically. Instead, it is also possible to implement support for session types as a library using only the existing type system of a language and lightweight dynamic checks. The library *Rusty Variation* [33] implements session types for the Rust language. It utilizes the type system of Rust to verify channels against session types at compile-time and guarantees deadlock freedom. Moreover, it can handle cancellation of sessions at runtime. There are also library implementations of session types for other languages (Haskell, Scala, OCaml) see [17]. Implementing a library-based implementation of session types is probably only possible for ABS by emulating channels at runtime and also performing all verification dynamically.

Typestate Typestate [45] which was introduced by Strom and Yemini is an extension of the type concept. Whereas a type defines the set of all operations applicable to a value in total, a typestate defines a subset of these operations permitted in a value's current context. This is achieved by associating values with an initial typestate in addition to their type and all operations with typestate transitions. Thus, our schedulers are a dynamic version of typestate since the subset of applicable methods of an active object is defined by the state and transitions of a Session Automaton. Originally, typestate was developed as a compile-time analysis but as Strom and Yemini point out, it becomes difficult to track typestate statically if values are shared among concurrent processes. This is the case with references to objects in ABS.

Hybrid Static and Dynamic Session Type Verification Some session type implementations which combine static and dynamic verification techniques (Session Java, Rusty Variation) have already been mentioned. They have in common that they use dynamic techniques to complement static verification where a compile-time solution is too difficult or impossible. If they detect a violation of a session type at run-time, execution is aborted. Our implementation behaves similar in this regard in that we statically verify method local behavior and dynamically enforce method invocation orders because the arrival and activation order of messages can not always be statically guaranteed. However, we do not abort execution upon detecting a violation of order but correct it instead by delaying activations. Only the verification of postconditions can lead to run-time errors.

A different approach on combining static and dynamic verification of session types, called *Gradual Session Types*, is presented by Igarashi et al. [25]. They give the programmer control over which parts of a program are verified statically or dynamically, since their functional language "Gradual GV" permits to combine statically and dynamically typed program fragments. Source programs are translated into an internal representation which adds run-time check annotations for the dynamically typed sections. However, Igarashi et al. provide no implementation of "Gradual GV".

6.2. Future Work

Throughout this thesis, we pointed out opportunities for improvement of our tool and also some shortcomings. In this section, we collect such issues.

Lifting Modeling Restrictions Our static verification process imposes some restrictions on programmers. For example

- futures must be stored in fields of a specific name as specified in a session type, see explanation of rule Call in Section 3.4.3. Also, futures can not be communicated between objects via method calls, see the discussion in Section 5.2.
- in a model, there can only be one instance of a class implementing an actor of a session type (see Section 3.4.2.)
- …

To lift these restrictions and provide more freedom in designing ABS models, our static verification techniques must be enhanced. For example, they could be extended with (pointer) analyses to track which future variables are referencing which invocation.

Improving upon Liveness Properties Due to several issues, we can give no strong guarantees on the liveness of a session, see Section 3.6. For one, deadlock checks of ABS models could be integrated to strengthen the liveness guarantees, see note at the bottom of Section 3.6. Also, work of Kamburjan et al. in [29] who check whether a session type specification induces deadlocks could be incorporated. They derive causality graphs from session types and inspect them for cycles.

The translation of session automata into data-deterministic automata needs to be completed (Section 3.5.3) and the limitations on schedulers described in Section 3.5.7 be resolved, if possible. Alternatively, algorithms could be implemented which can identify session types that induce these issues. Moreover, method bodies could be checked for possible exceptions which can prematurely cancel sessions.

Expanding Condition Checking As the example in Section 5.2 made clear, there are some situations in which preconditions should be checked. Preconditions can be added to our session types and verified at run-time in a similar fashion as postconditions with little effort.

Runtime verification can only detect the violation of a condition if a model is tested with the right set of input values while static verification can detect them early and for the general case. We could therefore improve the verification of postconditions by attempting it at compile time. Then, run-time checks are performed only for those postconditions which can not be statically proven to hold. There exists a deductive verification tool for ABS, KeY-ABS [14] which could be adapted to build such a feature.

Comprehensive Case Study Though we did evaluate our tool on some examples, see Chapter 5, applying it to an extensive ABS model which realizes complex communication protocols could lead to additional insights. For example, it could be examined to which degree the aforementioned restrictions on modeling hinder the implementation of the model. Also, it could be tested if the observations we recorded during our performance evaluation still hold for such a large-scale model or whether new effects of our schedulers on the execution performance of a model are discovered.

Bibliography

[1] Gul A Agha. *Actors: A model of concurrent computation in distributed systems.* Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[2] Davide Ancona et al. “Behavioral Types in Programming Languages”. In: *Foundations and Trends in Programming Languages* 3.2-3 (2016), pp. 95–230. DOI: 10.1561/2500000031. URL: https://doi.org/10.1561/2500000031.

[3] Pedro Baltazar, Dimitris Mostrous, and Vasco Thudichum Vasconcelos. “Linearly Refined Session Types”. In: *Proceedings 2nd International Workshop on Linearity, LINEARITY 2012, Tallinn, Estonia, 1 April 2012.* 2012, pp. 38–49. DOI: 10.4204/EPTCS.101.4. URL: https://doi.org/10.4204/EPTCS.101.4.

[4] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis”. In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 2007, pp. 504–518. DOI: 10.1007/978-3-540-73368-3_51. URL: https://doi.org/10.1007/978-3-540-73368-3%5C_51.

[5] Joakim Bjørk et al. “User-defined schedulers for real-time concurrent objects”. In: *Innovations in Systems and Software Engineering* 9.1 (Mar. 2013), pp. 29–43. ISSN: 1614-5054. DOI: 10.1007/s11334-012-0184-5. URL: https://doi.org/10.1007/s11334-012-0184-5.

[6] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. “A Complete Guide to the Future”. In: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. 2007, pp. 316–330. DOI: 10.1007/978-3-540-71316-6_22. URL: https://doi.org/10.1007/978-3-540-71316-6%5C_22.

[7] Frank S. de Boer et al. “A Survey of Active Object Languages”. In: *ACM Comput. Surv.* 50.5 (2017), 76:1–76:39. DOI: 10.1145/3122848. URL: https://doi.org/10.1145/3122848.

[8] Benedikt Bollig et al. “A Fresh Approach to Learning Register Automata”. In: *International Conference on Developments in Language Theory*. Ed. by Marie-Pierre Béal and Olivier Carton. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 118–130. ISBN: 978-3-642-38771-5.

[9] Joana Campos and Vasco T. Vasconcelos. “Channels as Objects in Concurrent Object-Oriented Programming”. In: *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010.* 2010, pp. 12–28. DOI: 10.4204/EPTCS.69.2. URL: https://doi.org/10.4204/EPTCS.69.2.

[10] Bo-Shoe Chen and Raymond T. Yeh. “Formal Specification and Verification of Distributed Systems”. In: *IEEE Trans. Software Eng.* 9.6 (1983), pp. 710–722. DOI: 10.1109/TSE.1983.235434. URL: https://doi.org/10.1109/TSE.1983.235434.

[11] Dave Clarke et al. *Full ABS Modeling Framework, Deliverable 1.2 of project FP7-231620 (HATS)*. Mar. 2011. URL: https://www.hats-project.eu.

[14] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. “KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS”. In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 2015, pp. 517–526. DOI: 10.1007/978-3-319-21401-6_35. URL: https://doi.org/10.1007/978-3-319-21401-6%5C_35.

[15] Cormac Flanagan and Matthias Felleisen. “The Semantics of Future and an Application”. In: *J. Funct. Program.* 9.1 (1999), pp. 1–31. URL: http://journals.cambridge.org/action/displayAbstract?aid=44231.

[16] Simon Fowler. “An Erlang Implementation of Multiparty Session Actors”. In: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.* 2016, pp. 36–50. DOI: 10.4204/EPTCS.223.3. URL: https://doi.org/10.4204/EPTCS.223.3.

[18] Juliana Franco and Vasco Thudichum Vasconcelos. “A Concurrent Programming Language with Refined Session Types”. In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*. 2013,

pp. 15–28. doi: 10.1007/978-3-319-05032-4_2. url: https://doi.org/10.1007/978-3-319-05032-4%5C_2.

[19] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. “A framework for deadlock detection in core□ABS”. In: *Software & Systems Modeling* 15.4 (Oct. 2016), pp. 1013–1048. issn: 1619-1374. doi: 10.1007/s10270-014-0444-y. url: https://doi.org/10.1007/s10270-014-0444-y.

[20] Reiner Hähnle. “The Abstract Behavioral Specification Language: A Tutorial Introduction”. In: *Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures*. 2012, pp. 1–37. doi: 10.1007/978-3-642-40615-7_1. url: https://doi.org/10.1007/978-3-642-40615-7%5C_1.

[21] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty asynchronous session types”. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 2008, pp. 273–284. doi: 10.1145/1328438.1328472. url: https://doi.org/10.1145/1328438.1328472.

[22] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. “Introduction to automata theory, languages, and computation, 3rd Edition”. In: Pearson international edition. Section: Equivalence of Deterministic and Nondeterministic Finite Automata. Addison-Wesley, 2007. Chap. 2, pp. 60–62. isbn: 978-0-321-47617-3.

[23] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. “Introduction to automata theory, languages, and computation, 3rd Edition”. In: Pearson international edition. Section: Finite Automata With Epsilon-Transitions. Addison-Wesley, 2007. Chap. 2, pp. 72–79. isbn: 978-0-321-47617-3.

[24] Raymond Hu, Nobuko Yoshida, and Kohei Honda. “Session-Based Distributed Programming in Java”. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. 2008, pp. 516–541. doi: 10.1007/978-3-540-70592-5_22. url: https://doi.org/10.1007/978-3-540-70592-5%5C_22.

[25] Atsushi Igarashi et al. “Gradual Session Types”. In: *CoRR* abs/1809.05649 (2018). arXiv: 1809.05649. url: http://arxiv.org/abs/1809.05649.

[27] Einar Broch Johnsen et al. “ABS: A Core Language for Abstract Behavioral Specification”. In: *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*.

2010, pp. 142–164. DOI: 10.1007/978-3-642-25271-6_8. URL: https://doi.org/10.1007/978-3-642-25271-6%5C_8.

[28] Eduard Kamburjan. *Session types for ABS*. Tech. rep. Technical report, 2016. URL: http://tubiblio.ulb.tu-darmstadt.de/105068/.

[29] Eduard Kamburjan and Tzu-Chun Chen. "Stateful Behavioral Types for Active Objects". In: *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*. 2018, pp. 214–235. DOI: 10.1007/978-3-319-98938-9_13. URL: https://doi.org/10.1007/978-3-319-98938-9%5C_13.

[30] Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. "Session-Based Compositional Analysis for Actor-Based Languages Using Futures". In: *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*. 2016, pp. 296–312. DOI: 10.1007/978-3-319-47846-3_19. URL: https://doi.org/10.1007/978-3-319-47846-3%5C_19.

[31] Michael Kaminski and Nissim Francez. "Finite-Memory Automata". In: *Theoretical Computer Science* 134.2 (1994), pp. 329–363. DOI: 10.1016/0304-3975(94)90242-9. URL: https://doi.org/10.1016/0304-3975(94)90242-9.

[33] Wen Kokke. "Rusty Variation: Deadlock-free Sessions with Failure in Rust". In: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*. 2019, pp. 48–60. DOI: 10.4204/EPTCS.304.4. URL: https://doi.org/10.4204/EPTCS.304.4.

[35] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710.

[36] Sam Lindley and J Garrett Morris. "Lightweight functional session types". In: *Behavioural Types: from Theory to Tools. River Publishers* (2017), pp. 265–286.

[38] Rumyana Neykova and Nobuko Yoshida. "Multiparty Session Actors". In: *Logical Methods in Computer Science* 13.1 (2017). DOI: 10.23638/LMCS-13(1:17)2017. URL: https://doi.org/10.23638/LMCS-13(1:17)2017.

[39] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. "Multiparty Session C: Safe Parallel Programming with Message Optimisation". In: *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*. 2012, pp. 202–218. DOI: 10.1007/978-3-642-30561-0_15. URL: https://doi.org/10.1007/978-3-642-30561-0%5C_15.

[45] Robert E. Strom and Shaula Yemini. “Typestate: A Programming Language Concept for Enhancing Software Reliability”. In: *IEEE Trans. Software Eng.* 12.1 (1986), pp. 157–171. DOI: 10.1109/TSE.1986.6312929. URL: https://doi.org/10.1109/TSE.1986.6312929.

[46] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. “An Interaction-based Language and its Typing System”. In: *PARLE ’94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*. 1994, pp. 398–413. DOI: 10.1007/3-540-58184-7_118. URL: https://doi.org/10.1007/3-540-58184-7%5C_118.

[47] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

Web Links

[12] JUnit Contributors. *JUnit 5 testing framework*. Accessed: 2019-10-29. URL: https://junit.org/junit5/.

[13] Linux kernel contributors. *Perf Tools*. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2019-10-24.

[17] Simon Fowler. *Session Types in Programming Languages: A Collection of Implementations*. Accessed: 2019-11-03. URL: http://groups.inf.ed.ac.uk/abcd/session-implementations.html.

[26] JetBrains s. r. o. and Kotlin Foundation. *The Kotlin Programming Language*. Accessed: 2019-10-29. URL: https://kotlinlang.org/.

[32] David Keppel et al. *GNU Time*. https://www.gnu.org/software/time/. Accessed: 2019-10-24.

[34] Ericsson Computer Science Laboratory. *The Erlang Programming Language*. Accessed: 2019-11-07. URL: https://www.erlang.org/.

[37] Commons IO Members and Contributors. *Commons IO utility library*. Accessed: 2019-10-29. URL: https://commons.apache.org/io/.

[40] Terence Parr and ANTLR Contributors. *ANTLR parser generator*. Accessed: 2019-10-29. URL: https://www.antlr.org/.

[41] Remko Popma and picocli Contributors. *picocli - a mighty tiny command line interface*. Accessed: 2019-10-29. URL: https://github.com/remkop/picocli.

[42] Armin Ronacher and Jinja contributors. *Jinja2 template engine*. https://jinja.palletsprojects.com/en/2.10.x/. Accessed: 2019-08-20.

[43] Rudolf Schlatte and abstools Contributors. *Our modified branch of the abstools compiler - GitHub source repository*. Accessed: 2019-10-29. URL: https://github.com/ahbnr/abstools/tree/thisDestiny.

[44] Rudolf Schlatte and abstools Contributors. *The ABS modeling language and surrounding tools - GitHub source repository*. Accessed: 2019-10-01. URL: https://github.com/abstools/abstools.

[48] Brett Wooldridge and NuProcess Contributors. *NuProcess. Low-overhead, non-blocking I/O, external Process implementation for Java*. Accessed: 2019-10-29. URL: https://github.com/brettwooldridge/NuProcess.

A. Tool Usage Instructions

The source code of the SDS-tool and a compiled version are distributed with this thesis on a compact disc which is attached to the last page. Moreover, the same source code can be downloaded from GitHub, though these links might be inaccessible until this thesis has been evaluated:

Our modified version of abstools `https://github.com/ahbnr/abstools`
(please use the branch “`thisDestiny`”)

SDS-tool `https://github.com/ahbnr/SessionTypeABS`

Prerequisites

Please make sure that the following dependencies are available on your system:

- Erlang version 22.1
- OpenJDK 11

If you want to build the tool from source, you also need to install the following dependencies:

- Kotlin compiler, version 1.3.50.
- Git, version 2.24.0

Furthermore, it is assumed, that the commands in the following subsections are executed on the `bash` shell of a linux system.

Instructions for building the SDS-tool can be found in the `README.md` file of the source distribution.

Usage

The following command

- statically verifies an ABS model against the given session types.
- applies our dynamic enforcement techniques to it.
- compiles the modified model to Erlang.

```
./sdstool compile [flags] [.abs files] [.st files]
```

With the optional flags, verification or dynamic enforcement can be deactivated etc. Use `./sdstool compile --help` for further information on them.

After compiling the ABS model, it can be executed like this:

```
gen/erl/run
```

Additional Commands

- `./sdstool printModel [.abs files] [.st files]` prints the parts of the given ABS model which are modified by our dynamic enforcement methods.
- `./sdstool printGlobalTypes [.st files]` parses the given session type files and outputs them again.
- `./sdstool printLocalTypes [.st files]` prints the object local session types projected from the given global ones.

B. Setup for Experiments

The experiments of Chapter 5 have been conducted on the following system configuration:

OS Arch Linux x86_64 (btw)

Kernel 5.3.7

CPU Intel i5-4300U (4) @ 2.900GHz

Memory 4 GiB

Execution time is measured using the *perf-stat* tool version 5.3.g4d856f72c10e [13] and the maximum memory resident set size is measured using *GNU Time* version 1.9-2 [32]. Different instances of the ABS models are automatically generated using the template engine Jinja2 [42]. The models can be executed and measurements be performed using a set of Python scripts in folder `prebuilt/evaluation` on the data medium distributed with this thesis. See the `README.md` / `README.pdf` file in the same folder for more instructions on how to run them.

For instructions on how to use the SDS-tool in general, see Appendix A.

C. Plot Data

In this section we give the data used for rendering the plots of Section 5.4 in tabular format.

repetitions	plain [s]	enforcement [s]
1	0.22	0.21
2	0.2	0.2
3	0.2	0.21
4	0.21	0.21
5	0.2	0.21
10	0.2	0.21
30	0.22	0.23
50	0.22	0.22
70	0.23	0.25
80	0.25	0.26
100	0.24	0.28
300	0.34	0.59
500	0.44	1.02
700	0.51	1.45
900	0.58	2.06

repetitions	relative increase [%]
1	−6.37
2	−2.04
3	3.62
4	1.53
5	7.24
10	5.63
30	5.92
50	0.59
70	6.66
80	6.56
100	14.7
300	73.41
500	132.51
700	184.65
900	252.58

Table C.1.: User-mode execution time. Left table displays absolute values, right one relative increase in time when using Dynamic Enforcement.

repetitions	plain [kB]	enforcement [kB]
1	26,272	25,983.6
2	26,074.8	26,291.6
3	25,896.4	26,210.8
4	25,934.8	26,262.8
5	26,318	26,289.2
10	25,934.4	26,175.2
30	26,200	26,319.6
50	26,349.6	27,148.8
70	26,798.8	26,623.6
80	26,724.8	27,092.4
100	27,280.4	27,311.6
300	29,475.6	30,913.6
500	31,906.8	35,234.4
700	34,628	40,526.4
900	36,952	45,111.2

repetitions	relative increase [%]
1	-1.1
2	0.83
3	1.21
4	1.26
5	-0.11
10	0.93
30	0.46
50	3.03
70	-0.65
80	1.38
100	0.11
300	4.88
500	10.43
700	17.03
900	22.08

Table C.2.: Maximum memory set size. Left table displays absolute values, right one relative increase in time when using Dynamic Enforcement.

repetitions	plain sequential [edits]	plain alternating [edits]	enforcement sequential [edits]	enforcement alternating [edits]	sequence length
1	0	0	0	0	2
2	0	0	0	0	4
3	0.4	0.4	0	0	6
4	0.6	0.6	0	0	8
5	0.6	0.6	0	0	10
10	5.2	3.6	0	0	20
30	28.6	14.75	0	0	60
50	60.1	28.3	0	0	100
70	94.45	41.7	0	0	140
80	103.5	45.65	0	0	160

repetitions	edits / sequence length
1	0
2	0
3	$6.67 \cdot 10^{-2}$
4	$7.5 \cdot 10^{-2}$
5	$6 \cdot 10^{-2}$
10	0.18
30	0.25
50	0.28
70	0.3
80	0.29

Table C.3.: Levenshtein distance of observed and expected invocation sequence. Absolute edits are displayed in the upper table. In the lower table, the number of edits per sequence length is displayed when expecting a plain alternating sequence.

repetitions	delays	calls of scheduler
1	0	3
2	0	5
3	0	7
4	0	9
5	0	11
10	0	21
30	0	61
50	0	101
70	0	141
80	0	161
100	0	201
300	0	601
500	0	1,001
700	0	1,401
900	0	1,801

Table C.4.: Calls of the scheduler of Q. A “delay” is counted for every time the scheduler is called and it can not select an activation.

repetitions	plain [s]	enforcement [s]
1	0.2	0.2
2	0.2	0.2
3	0.21	0.21
4	0.22	0.21
5	0.21	0.21
10	0.22	0.22
30	0.23	0.23
50	0.25	0.26
70	0.26	0.29
80	0.27	0.29
100	0.31	0.31
300	0.45	0.47
500	0.61	0.68
700	0.76	0.81
900	0.92	0.99

repetitions	relative increase [%]
1	0.25
2	0.41
3	2.53
4	−4.65
5	−2.68
10	0.26
30	0.66
50	1.86
70	13.06
80	5.74
100	0.95
300	4.77
500	11.85
700	6.07
900	7.51

Table C.5.: User-mode execution time when using await statements. Left plot displays absolute values, right one relative increase in time when using Dynamic Enforcement.

repetitions	delays	calls of scheduler
1	1	4
2	2	7
3	3	10
4	4	13
5	5	16
10	10	31
30	30	91
50	50	151
70	70	211
80	80	241
100	100	301
300	300	901
500	500	1,501
700	700	2,101
900	900	2,701

repetitions	plain	enforcement
	[s]	[s]
1	0.2	0.2
2	0.2	0.2
3	0.2	0.24
4	0.24	0.28
5	0.21	0.21
10	0.26	0.28
30	0.25	0.24
50	0.27	0.27
70	0.28	0.3
80	0.31	0.32
100	0.33	0.35
300	0.74	0.71
500	1.17	1.25
700	1.91	1.9
900	2.65	2.72

Table C.6.: The left table shows the number of times ("delays") no activation could be selected by the scheduler of Q compared to the total number of calls. The right table depicts measured user-mode execution times.

repetitions	plain [s]	enforcement [s]
1	0.2	0.21
2	0.21	0.21
3	0.2	0.2
4	0.22	0.21
5	0.2	0.2
10	0.21	0.22
30	0.23	0.23
50	0.23	0.23
70	0.24	0.25
80	0.24	0.26
100	0.26	0.26
300	0.35	0.37
500	0.45	0.46
700	0.55	0.56
900	0.63	0.67

repetitions	relative increase [%]
1	$3.72 \cdot 10^{-2}$
2	$-1.66 \cdot 10^{-2}$
3	$-2.79 \cdot 10^{-3}$
4	$-2.6 \cdot 10^{-3}$
5	$-6.67 \cdot 10^{-4}$
10	$6.06 \cdot 10^{-2}$
30	$-8.22 \cdot 10^{-3}$
50	$1.31 \cdot 10^{-2}$
70	$1.97 \cdot 10^{-2}$
80	$6.78 \cdot 10^{-2}$
100	$6.79 \cdot 10^{-3}$
300	$6.38 \cdot 10^{-2}$
500	$2.22 \cdot 10^{-2}$
700	$1.96 \cdot 10^{-2}$
900	$6.67 \cdot 10^{-2}$

repetitions	plain [kB]	enforcement [kB]
1	26,334.8	26,406.8
2	26,358.8	25,880
3	26,309.2	25,962
4	26,219.6	26,295.6
5	26,170.4	25,980.4
10	26,056.8	26,424.4
30	26,179.2	26,550.4
50	26,554	26,800.4
70	26,723.6	26,903.6
80	26,938	27,340.8
100	27,131.2	27,484.8
300	29,114.8	29,273.6
500	31,268.8	31,857.6
700	33,781.6	34,030
900	36,610.8	36,880.4

repetitions	relative increase [%]
1	$2.73 \cdot 10^{-3}$
2	$-1.82 \cdot 10^{-2}$
3	$-1.32 \cdot 10^{-2}$
4	$2.9 \cdot 10^{-3}$
5	$-7.26 \cdot 10^{-3}$
10	$1.41 \cdot 10^{-2}$
30	$1.42 \cdot 10^{-2}$
50	$9.28 \cdot 10^{-3}$
70	$6.74 \cdot 10^{-3}$
80	$1.5 \cdot 10^{-2}$
100	$1.3 \cdot 10^{-2}$
300	$5.45 \cdot 10^{-3}$
500	$1.88 \cdot 10^{-2}$
700	$7.35 \cdot 10^{-3}$
900	$7.36 \cdot 10^{-3}$

Table C.7.: In the upper row: User-mode execution time. In the lower row: Maximum memory set size. The left tables display absolute values, the right ones the relative increase in time/memory when using Dynamic Enforcement.

D. Algorithms

Algorithm 2 ε-NFA to DFA algorithm based on the powerset construction.

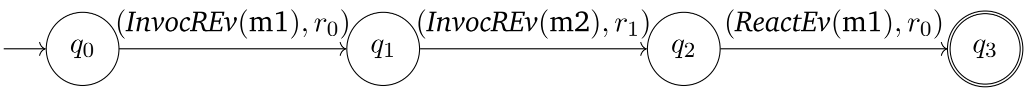
```
procedure $\varepsilon$NFAToDFA$(Q, q_0, \Delta, F)$
    $\overline{q}_0 := \varepsilon\textsc{Closure}(q_0)$
    $\overline{Q} := \{\overline{q}_0\}$
    $\overline{\Delta} := \emptyset$
    $\overline{F} := \emptyset$

    $\mathit{Inspected} := \emptyset$

    while there is an $\overline{q} \in \overline{Q}$ with $\overline{q} \notin \mathit{Inspected}$ do
        $\overline{\Delta}_+ := \textsc{groupedTransitions}(\overline{q})$
        $\overline{Q} \leftarrow \overline{Q} \cup \{\overline{q}_2 \mid (\overline{q}_1, v, \overline{q}_2) \in \overline{\Delta}_+\}$
        $\overline{\Delta} \leftarrow \overline{\Delta} \cup \overline{\Delta}_+$

        $\mathit{Inspected} \leftarrow \mathit{Inspected} \cup \{\overline{q}\}$

    $\overline{F} = \{\overline{q} \in \overline{Q} \mid \exists q \in \overline{q}.\, q \in F\}$
    return $(\overline{Q}, \overline{q}_0, \overline{\Delta}, \overline{F})$
function $\textsc{groupedTransitions}(\overline{q})$
    return $\{(\overline{q}, v, \bigcup_{(q_1,v,q_2)\in\Delta} \varepsilon\textsc{Closure}(q_2)) \mid q_1 \in \overline{q} \wedge (q_1, v, \cdot) \in \Delta \wedge v \neq \varepsilon\}$
procedure $\varepsilon\textsc{Closure}(q)$
    $\mathit{Closure} := \emptyset$
    $\mathit{ToProcess} := \{q\}$

    while there is $q_1 \in \mathit{ToProcess}$ do
        $\mathit{Closure}_+ := (\{q_1\} \cup \{q_2 \mid (q_1, \varepsilon, q_2) \in \Delta\}) \setminus \mathit{Closure}$
        $\mathit{Closure} \leftarrow \mathit{Closure} \cup \mathit{Closure}_+$

        $\mathit{ToProcess} \leftarrow (\mathit{ToProcess} \cup \mathit{Closure}_+) \setminus \{q_1\}$

    return $\mathit{Closure}$
```

E. Examples

Example 8: AST Modifications for Enforcement of a Scheduling Policy

The following automaton A describes a scheduling policy for a class C, where

1. a method m1 can be invoked
2. a method m2 can be invoked
3. the computation of the first invocation can be reactivated


q_0
$(\mathit{InvocREv}(\mathrm{m1}), r_0)$
q_1
$(\mathit{InvocREv}(\mathrm{m2}), r_1)$
q_2
$(\mathit{ReactEv}(\mathrm{m1}), r_0)$
q_3

Class C is defined like this:

```
class C (DI d) implements CI {
  Fut<Unit> f;

  Unit m1() {
    this.f = d!m();
    await this.f?;
  }

  Unit m2() {
    //...
  }
}
```

Then this is the result of applying all necessary modifications to enforce the scheduling policy described by A:

```
fun schedule(
  List<Process> queue,
  Int q,
  Register r1, Register r2
) = forceInit(
  () => case q {
    0 => headOrNothing(
      filter((Process p) =>
              !contains(set[r1, r2], Just(destinyOf(p)))
           && contains(set["m1"], method(p))
        || contains(set[], Just(destinyOf(p)))
      )(queue)
    );
    1 => headOrNothing(
      filter((Process p) =>
              !contains(set[r1, r2], Just(destinyOf(p)))
           && contains(set["m2"], method(p))
        || contains(set[], Just(destinyOf(p)))
      )(queue)
    );
    2 => headOrNothing(
      filter((Process p) =>
              !contains(set[r1, r2], Just(destinyOf(p)))
           && contains(set[], method(p))
        || contains(set[r1], Just(destinyOf(p)))
      )(queue)
    );
    3 => headOrNothing(
      filter((Process p) =>
              !contains(set[r1, r2], Just(destinyOf(p)))
           && contains(set[], method(p))
        || contains(set[], Just(destinyOf(p)))
      )(queue)
    );
  }
)(queue);
```

```
[Scheduler: schedule(queue, q, r1, r2)
class C (DI d) implements CI {
  Int q = 0;
  Register r1 = Nothing;
  Register r2 = Nothing;

  Fut<Unit> f;

  Unit m1() {
    case this.q {
      0 => {
        this.r1 = destiny;
        this.q = 1;
      }
      _ => assert False;
    }

    this.f = d!m();
    await this.f?;
    if (this.q == 2 && this.r1 == Just(destiny)) {
      this.q = 3;
    }

    else { assert False; }
  }

  Unit m2() {
    case this.q {
      1 => {
        this.r2 = destiny;
        this.q = 2;
      }
      _ => assert False;
    }
    //...
  }
}
```

F. Code Listings

```
...
Unit start(Int v){
  Int invocState = this.q;
  case this.q {
    0 => { this.r0 = Just( destiny ); this.q = 1; }
    _ => { assert False; }
  }

  this.fCmp = this.b!cmp(this, v);
  this.intern = Expect;

  case invocState {
    0 => {
      assert ( this.intern == Expect );
    }
    _ => skip;
  }
}
...
```

Listing 11: Generated AST modifications of method start of the ABS model of Section 5.2.

```
0 –f–> P:m.
(
  P –fm1–> Q:m1.
    Q resolves fm1.
  P –fm2–> Q:m2.
    Q resolves fm2
)*.
P resolves f
```

Listing 12: Source code for the session type of the performance evaluation model of Section 5.4.

G. Miscellaneous

G.1. Allowed Pure Expressions In Postconditions

The SDS-tool supports the following kinds of ABS pure expressions as defined in the ANTLR grammar of the *abstools* compiler for the postconditions introduced in Section 3.7:

- FunctionExp
- ConstructorExp
- UnaryExp
- MultExp
- AddExp
- GreaterExp
- EqualExp
- AndExp
- OrExp
- VarOrFieldExp
- FloatExp
- IntExp
- StringExp
- ThisExp
- DestinyExp
- NullExp
- ParenExp

G.2. Used Libraries and Tools

Implementation

- our modified variation of abstools version 1.8.1 [44, 43]
- ANTLR version 4.7 [40]
- picocli version 4.0.0-beta-2 [41]
- Apache Commons IO version 2.6 [37]

Testing

- JUnit 5 version 5.5.0 [12]
- NuProcess version 1.2.3 [48]

H. Index of Figures, Tables, Etc.

List of Figures

List of Tables

List of Theorems

List of Algorithms

List of Examples