Session-Based Compositional Analysis for Actor-Based Languages Using Futures ⋆

Eduard Kamburjan[1], Crystal Chang Din[2], and Tzu-Chun Chen[1]

[1] Department of Computer Science, TU Darmstadt, Germany
[2] Department of Informatics, University of Oslo, Norway
kamburjan@cs.tu-darmstadt.de, crystald@ifi.uio.no,
tc.chen@dsp.tu-darmstadt.de

Abstract. This paper proposes a simple yet concise framework to statically verify communication correctness in a concurrency model using futures. We consider the concurrency model of the core ABS language, which supports actor-style asynchronous communication using futures and cooperative scheduling. We provide a type discipline based on session types, which gives a high-level abstraction for structured interactions. By using it we statically verify if the local implementations comply with the communication correctness. We extend core ABS with sessions and annotations to express scheduling policies based on required communication ordering. The annotation is statically checked against the session automata derived from the session types.

1 Introduction

While distributed and concurrent systems are the pillars of modern IT infrastructures, it is non-trivial to model asynchronous interactions and statically guarantee communication correctness of such systems. This challenge motivates us to bring a compositional analysis framework, which models and locally verifies the behaviors of each distributed endpoints (i.e. components) from the specification of their global interactions. For modeling, we focus on core ABS [10, 15], an object-oriented actor-based language designed to model distributed and concurrent systems with asynchronous communications. For verification, we establish a hybrid analysis, which statically type checks local objects' behaviors and, at the same time ensures that local schedulers obey to specified policies during runtime. We apply *session types* [12, 21] to type interactions by abstracting structured communications as a global specification, and then automatically generating local specifications from the global one to locally type check endpoint behaviors.

The distinguishing features of the core ABS concurrency model are (1) *cooperative scheduling*, where methods explicitly control internal interleavings by explicit scheduling points, and (2) the usage of *futures* [11], which decouple the process invoking a method and the process reading the returned value. By sharing future identities, the caller enables other objects to wait for the same method

⋆ Every author contributed to this paper equally.

results. Note that core ABS does not use channels. Communication between processes is restricted to method calls and return values.

The order of operations on futures is fixed: First a fresh future identity is created upon the caller invoking a method on the callee, then the callee starts the method execution. After method termination, the callee sends the result to the future, i.e. future is resolved. Finally, any object which can access the future can read the value from this future. Our session-based type system ensures that the specification respects this order. We have two kinds of communications: Caller invoking a method at a remote callee, and callee returning values via a future to those who know that future. The later is non-trivial since several endpoints can read more than once from the same resolved future at any time.

To the best of our knowledge, it is the first time that session types are considered for typing the concurrency model of core ABS. Our contributions include: (1) extending core ABS with sessions (SABS for short) by giving special annotations to specify the order of interactions among concurrent class instances, (2) establishing a session-based type system, and (3) generating session automata [2] from session types to represent scheduling policies and typestate [20]. To capture the interactions among objects, which are running several processes, we introduce a *two-fold notion of local types*: Object types defining behaviors (i.e. including scheduling behavior) among class instances, while method types defining behaviors that processes should follow.

Outline: Section 2 gives a motivating example which is used in the rest of the paper. Section 3 introduces the concurrency model of SABS. Section 4 defines session types for SABS (ABS-ST for short), while Section 5 gives a type system. Section 6 introduces session automata which are used to verify behaviors of schedulers. Section 7 gives the related works, while section 8 concludes our work.

2 Motivating Example: A Grading System

Consider a service, called *grading system*, which offers an expensive computation on sensitive data, e.g. automatic evaluation of exams. This service consists of three endpoints: A computation server, denoted by c, and a service desk, denoted by d, where a student, denoted by s, can request their grades. The protocol is as follows: Once c finishes calculating the grades, it sends a *publish* message containing the grades to d and an announcement, *announce* message, to a student. It is not desirable that d starts a new communication with c because c may be already computing the next exams; it is also not desirable that d communicates to s without any request from s.

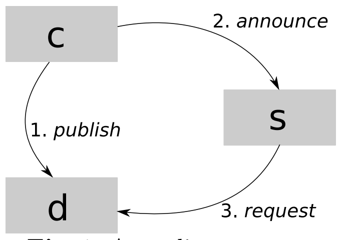

Fig. 1. A grading system

If a student requests his/her grades before the service desk receives the grades from c by *publish*, the scheduler of d must postpone the process of *request* until *publish* has been executed and terminated. This is not possible for core ABS, because a scheduler in core ABS cannot be idle while waiting for a specific message when the process queue is non-empty. Thus we propose an extension of core ABS to ensure that the endpoints and their local schedulers behave well to the specified communication order.

3 The Session-based ABS Language (SABS)

This section introduces the concept of *session* to core ABS [10, 15]. The extended language is called session-based ABS, SABS in short. The goal of this extension is to equip the language's compiler with the ability to statically ensure communication correctness for applications written in core ABS.

3.1 Syntax and the concurrency model of core ABS

The SABS language provides a combination of algebraic datatype, functional sublanguage, and a simple imperative object-oriented language. The former two kinds are kept the same in SABS as in core ABS. The imperative object-oriented layer is extended. The syntax of SABS can be found in Fig. 2, in which the new language extension is highlighted and will be explained in Section 3.2.

Syntactic categories	*Definitions*
T in Ground Type	$T ::= B \mid I \mid D \mid D\langle\overline{T}\rangle \quad B ::= \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{String} \mid \mathsf{Unit} \mid \cdots$
B in Basic Type	$A ::= N \mid T \mid D\langle\overline{A}\rangle$
A in Generic Type	$Dd ::= \mathbf{data}\ D[\langle\overline{A}\rangle] = Cons[\overline{\mid Cons}]; \qquad Cons ::= Co[(\overline{A})]$
N in Names	$F ::= \mathbf{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\ \overline{x}) = e;$
e in Expression	$e ::= b \mid x \mid t \mid \mathbf{this} \mid \mathbf{destiny} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \mathbf{case}\ e\ \{\overline{br}\}$
x in Variable	$t ::= Co[(\overline{t})] \mid \mathbf{null} \quad br ::= p \Rightarrow e; \quad p ::= \_ \mid x \mid t \mid Co[(\overline{p})]$
t in Ground Term	$P ::= \overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x};\ s\}$
br in Branch	$IF ::= \mathbf{interface}\ I\ \{\ \overline{Sg}\ \} \qquad Sg ::= T\ m\ (\overline{T}\ \overline{x})$
p in Pattern	$CL ::= [[a]]\mathbf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\mathbf{implements}\ \overline{I}]\ \{\ \overline{T}\ \overline{x};\ \overline{M}\}$
C, I, m in Names	$M ::= Sg\ \{\ \overline{T}\ \overline{x};\ s\ \}$
$\mathbf{L}$ in Local Types	$a ::= \mathbf{Scheduler\colon L} \quad ptc ::= \mathbf{Protocol\colon G} \quad join ::= \mathbf{Ses}\colon \mathsf{String}$
S in Session IDs	$rhs ::= e \mid e!m(\overline{e}) \mid e.\mathbf{get}$
$\mathbf{G}$ in Global Types	$s ::= s; s \mid x = rhs \mid \mathbf{skip} \mid \mathbf{return}\ e$
s in Statement	$\mid \mathbf{await}\ e? \mid \mathbf{if}\ b\ \{s\}\ [\mathbf{else}\ \{s\}] \mid \mathbf{while}\ b\ \{s\} \mid \mathbf{case}\ e\{\overline{p \Rightarrow s;}\}$
b in Bool Expression	$\mid [ptc]\ \mathrm{x} = \mathbf{new}\ Session(\mathrm{e}) \mid [join]\ x = \mathbf{new}\ C\,[(\overline{e})]$

Fig. 2. SABS syntax. Terms $\bar{\cdot}$ denote possibly empty lists over corresponding syntactic categories, and [] optional elements. The highlighted ones are the new syntax added to core ABS.

A SABS model, denoted by P, defines datatypes $\overline{Dd}$, functions $\overline{F}$, interfaces $\overline{IF}$, classes $\overline{CL}$, and main block $\{\overline{T}\ \overline{x};\ \mathrm{s}\}$ to configure the initial state. In datatype declarations Dd, an abstract datatype D has at least one constructor $Cons$, which has a name Co and a list of generic types $\overline{A}$ for its arguments. A future of built-in type $\mathbf{Fut}\langle T\rangle$ expresses that the value stored in the future is of type T. Function declarations F consist of a return type A, a function name fn, an optional list of parameters of types $\overline{A}$, a list of variable declarations $\overline{x}$ of types $\overline{A}$, and an expression e. Expressions e include boolean expressions b, variables x, (ground) terms t, the self-identifier **this**, the return address **destiny** of the method activation, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$ where br is a branch. An interface $\overline{IF}$

has a name I and method signatures $\overline{Sg}$. A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. A class CL has a name C, the fields $\overline{x}$ of type $\overline{T}$ for formal parameters and state variables, implemented interfaces $\overline{I}$ and methods $\overline{M}$. The right-hand side expressions *rhs* include (pure) expressions e, asynchronous remote method invocation $e!m(\overline{e})$, and future fetching expression $e.$**get**. Statements s include sequential composition, assignment, session creation, object creation, guarding statement **await** $e?$, **if**, **while**, branching, **skip** and **return** statement.

The concurrency model In SABS each object has one scheduler and one processor. It is possible to have more than one processes on an object, but at most one process is executed by the processor on an object at a time. For a method call, a fresh future identity, say f, is generated by the caller upon sending an asynchronous remote method invocation to the callee. A future can be seen as a placeholder for the method result. The callee creates a new process for the receiving call. If the processor of the callee is busy while the new message arrives, the created process will be put into the process pool and can later be chosen for execution by the scheduler. Upon method termination, the callee returns the result to f, i.e. f is resolved. Any object sharing the identity f can read the value from f by executing $f.$**get**. This statement blocks the current process until f is resolved and then returns the value. Since execution control is not transferred between objects and there is no direct access from one object to the fields of other objects, each object can be analyzed individually. SABS supports cooperative scheduling. Each object relies on its scheduler to select a process for execution at the explicit scheduling points, which can be upon termination of object initialization, at **await** statement, and upon method termination. When a process execution encounters statement **await** $f?$, if the future f is not resolved yet, the processor is released and the current process is suspended and put into the process pool. Then the processor is idle and the scheduler chooses a process from the pool for execution based on a scheduling policy (i.e. specified by a local specification). We say the chosen process from the pool is reactivated.

3.2 New language extension

SABS provides a set of new features in order to guide the scheduler to select the intended process for execution according to the required interaction ordering. In Fig. 2, the statement **[Protocol : G]** $x =$ **new** *Session*(e) creates a new session with a fresh session *id* stored in x. The parameter e is the session name of type *String*. The annotation **[Protocol : G]** describes the global communication specification **G**, which will be formalized in Section. 4.1, that the newly created session should obey. The statement **[Ses :** S**]** $x =$ **new** $C\,[(\overline{e})]$ creates a new object with a fresh object *id* stored in x. The annotation **[Ses :** S**]** specifies that the newly created object belongs to session S. Each object can belong to at most one session. The annotation **[Scheduler : L]** is optional and can be added in front of the class declarations. It provides the local communication specification **L** to guide the scheduler of the current object.

The SABS implementation for the grading example in Section 2 is in Fig. 3, in which we create a new session ses named Service. Type gradingSystem de-

```
interface ServiceDesk{ Unit publish(Int g); Int request(); }
interface CompServer{ Unit pubGrd(ServiceDesk d, Student s, Int g); }
interface Student{ Unit announce(ServiceDesk d); }

[ Scheduler: c?$_f$publish.s?$_{f'}$request] // local protocol for SD
class SD(CompServer c, Student s) implements ServiceDesk{
          Int grade = 0;
          Unit publish(Int g){grade = g;}
          Int request(){return grade;}}

class CS implements CompServer{
          Unit pubGrd(ServiceDesk d, Student s, Int g){
                    Fut<Unit> f = d!publish(g,this);
                    Fut<Unit> f' = s!announce(d,this);
          }}

class S implements Student{
          Int grade = 0;
          Unit announce(ServiceDesk d){
                    Fut<Int> f'' = d!request(this);
                    grade = f''.get;
          }}

{ [ Protocol: gradingSystem ] Ses ses = new Session("Service");
   [ Ses: ses ] CompServer c = new CS;
   [ Ses: ses ] Student s = new S;
   [ Ses: ses ] ServiceDesk d = new SD(c,s);
  Fut<Unit> f0 = c!pubGrd(d,s,85); }
```

Fig. 3. The ABS implementation for the example in Section. 2.

fines this session's global communication specification (introduced later in Section 4.1). Computer server c, student s and service desk d all belong to the same session ses. The scheduling policy for the service desk is represented by a local specification $\mathsf{c}?_f \mathit{publish}.\mathsf{s}?_{f'}\mathit{request}$ (introduced later in Section 4.2), which specifies the method request invoked by the student s can only be executed after the execution of method publish invoked by the computer server c.

4 Compositional Analysis Based on Session Types

Based on the approach of compositional analysis and the theory of session types [12], we introduce the ABS-ST (*ABS Session Types*) framework: Each object (i.e. component) is statically checked against its local specification, which are projected from a global specification, specifying the overall interactions among objects (i.e. composes objects). As multiparty session types type interactions consisting of simple sending and receiving actions among multiple processes, ABS-ST type interactions consisting of asynchronous remote method calls, scheduling, and futures among *objects*. This work extends [16], which contains proofs, full examples and definitions.

4.1 Global Types

Global types, denoted by $\mathbf{G}$, define global communication specifications within a closed system of objects. Contrary to session types [12, 21], we do not specify the datatype of a message since the message is a method call or a method return and every method in SABS has a fixed signature. The syntax of $\mathbf{G}$ is defined:

Definition 1. Let $\mathbf{p}, \mathbf{q}$ range over objects, denoted by Ob, f over futures, m over method names and $\mathtt{C}$ over all constructors of all abstract datatypes.

$$\mathbf{G} ::= \mathbf{0} \xrightarrow{f} \mathbf{q}\!:\!m \mid \mathbf{G}.\mathbf{g} \qquad \begin{array}{c}\mathbf{g} ::= \mathbf{p} \xrightarrow{f} \mathbf{q}\!:\!m.\mathbf{g} \mid \mathbf{p}{\downarrow}f\!:\!(\mathtt{C}).\mathbf{g} \mid \mathbf{p}{\uparrow}f\!:\!(\mathtt{C}).\mathbf{g} \mid \\ \mathsf{Rel}(\mathbf{p}, f).\mathbf{g} \mid \mathbf{p}\{\mathbf{g}_j\}_{j\in J} \mid \mathsf{end} \mid \mathbf{g}^*\end{array}$$

Initialization $\mathbf{0} \xrightarrow{f} \mathbf{q}\!:\!m$ starts interactions from the *main block* invoking object $\mathbf{q}$, e.g. we write $\mathbf{0} \xrightarrow{f_0} \mathbf{c}\!:\!pubGrd$ to specify the code in the *main block* in Fig. 3. We use . for sequential composition and write $\mathbf{G}.\mathbf{g}$ to mean interaction(s) $\mathbf{g}$ follows $\mathbf{G}$. Interaction $\mathbf{p} \xrightarrow{f} \mathbf{q}\!:\!m$ models a remote call, where object $\mathbf{p}$ asynchronously calls method m at object $\mathbf{q}$ via future f, and then $\mathbf{q}$ creates a new process for this method call. The resolving type $\mathbf{p}{\downarrow}f\!:\!(\mathtt{C})$ models object $\mathbf{p}$ resolving the future f. If the method has an algebraic datatype as its return type, then the return value has $\mathtt{C}$ as its outermost constructor; otherwise we simply write $\mathbf{p}{\downarrow}f$. The fetching type $\mathbf{p}{\uparrow}f\!:\!(\mathtt{C})$ models object $\mathbf{p}$ reading the future f. The usage of $\mathtt{C}$ here is similar to the one in $\mathbf{p}{\downarrow}f\!:\!(\mathtt{C})$. The releasing type $\mathsf{Rel}(\mathbf{p}, f)$ models $\mathbf{p}$ which releases the control until future f has been resolved. This type corresponds to **await** f? statement in SABS. The example below shows how $\mathsf{Rel}(\mathbf{p}, f)$ works:

Example 1. Consider $\mathbf{G}_{\text{release}} = \mathbf{0} \xrightarrow{f_0} \mathbf{a}\!:\!m_0.\mathbf{a} \xrightarrow{f_1} \mathbf{b}\!:\!m_1.\mathbf{b} \xrightarrow{f_2} \mathbf{a}\!:\!m_2.\mathbf{g}$. It does not specify the usage of futures correctly: At the moment $\mathbf{b}$ makes a remote call on m_2 at $\mathbf{a}$, the process computing f_0 is still active at $\mathbf{a}$. We shall revise it to

$$\mathbf{G}'_{\text{release}} = \mathbf{0} \xrightarrow{f_0} \mathbf{a}\!:\!m_0.\mathbf{a} \xrightarrow{f_1} \mathbf{b}\!:\!m_1.\mathsf{Rel}(\mathbf{a}, f_1).\mathbf{b} \xrightarrow{f_2} \mathbf{a}\!:\!m_2.\mathbf{g}$$

Here $\mathbf{a}$ suspends its first process computing f_0 until f_1 has been resolved; during this period, $\mathbf{a}$ can execute the call on m_2.

The branching type $\mathbf{p}\{\mathbf{g}_j\}_{j\in J}$ expresses that as $\mathbf{p}$ selects the jth branch, $\mathbf{g}_j$ guides the continuing interactions. The type end means termination.

Note that only a *self-contained* $\mathbf{g}$ can be repeatedly used. We say $\mathbf{g}$ is *self-contained* if (1) wherever there is a remote call or releasing, there is a corresponding resolving and visa versa; and (2) it contains no end, and (3) every repeated type within it is also *self-contained.* We say $A \in \mathbf{g}$ if A appears in $\mathbf{g}$ and $A \in \mathbf{G}$ if $A \in \mathbf{g}$ for some $\mathbf{g} \in \mathbf{G}$. E.g., we have $\mathbf{q} \xrightarrow{f} \mathbf{p} : m \in \mathbf{0} \xrightarrow{f_0} \mathbf{q} : m.\mathbf{q} \xrightarrow{f} \mathbf{p} : m$ and $\mathbf{q}\{\mathbf{g}_2\} \in \mathbf{q}\{\mathbf{g}_j\}_{j\in\{1,2,3\}}$ and its negation means the inverse. A future f is *introduced in* $\mathbf{g}$ (or $\mathbf{G}$) if $\mathbf{p} \xrightarrow{f} \mathbf{q} \in \mathbf{g}$ (or $\mathbf{G}$).

Now we define type $\mathbf{g}^* = \mathit{fresh}(\mathbf{g}).\mathbf{g}^*$ (in case $\mathit{fresh}(\mathbf{g})$ is a branching, we append $\mathbf{g}^*$ to the end of every branch), meaning finite repetition of a *self-contained*

g by giving every repetition fresh future names:

$$fresh(\mathbf{g}) = \begin{cases} \mathbf{g}\{f_1'/f_1\}...\{f_n'/f_n\} & \text{if } f_1, ..., f_n \text{ are } \textit{introduced in } \mathbf{g} \text{ and } f_1', .., f_n' \text{ fresh} \\ \mathbf{p}\{fresh(\mathbf{g}_j)\}_{j\in J} & \text{if } \mathbf{p}\{\mathbf{g}_j\}_{j\in J} \\ \text{Undefined} & \text{otherwise} \end{cases}$$

In other words, we need to keep *linearity* of futures for every iterations.

Example 2. We show how the global type gradingSystem, used in the code of Fig. 3, represents the grading system discussed in Section 2:

$$\begin{aligned} \mathsf{gradingSystem} = \; & \mathbf{0} \xrightarrow{f_0} \mathbf{c}{:}\textit{pubGrd}.\mathbf{c} \xrightarrow{f} \mathbf{d}{:}\textit{publish}.\mathbf{d}{\downarrow} f.\mathbf{c} \xrightarrow{f'} \mathbf{s}{:}\textit{announce}. \\ & \mathbf{s} \xrightarrow{f''} \mathbf{d}{:}\textit{request}.\mathbf{d}{\downarrow} f''.\mathbf{s}{\uparrow} f''.\mathbf{s}{\downarrow} f'.\mathbf{c}{\downarrow} f_0.\mathsf{end} \end{aligned}$$

The session is started by a call on $\mathbf{c}.\textit{pubGrd}$, while other objects are inactive at the moment. After the call $\mathbf{c} \xrightarrow{f} \mathbf{d} : \textit{publish}$, the service desk **d** is active at computing f in a process running *publish*. We position $\mathbf{d} \downarrow f$ there to specify that **d** must resolve f after it is called by **c** and before it is called by **s** (i.e. $\mathbf{s} \xrightarrow{f''} \mathbf{d} : \textit{request}$). For **c**, it can have a second remote call $\mathbf{c} \xrightarrow{f'} \mathbf{s} : \textit{announce}$ after its first call. Thus in this case it is no harm to move $\mathbf{d} \downarrow f$ right after $\mathbf{c} \xrightarrow{f'} \mathbf{s} : \textit{announce}$. As **d** is called by **s**, **d** can start computing f'' in a process running *request* only after $\mathbf{d}{\downarrow} f$, which means the process computing *publish* has terminated. **s** will fetch the result by $\mathbf{s}{\uparrow} f''$ after **d** resolves f''; then **s** resolves f'. Note that, since **c** does not need to get any response from **d** nor **s**, **c** simply finishes the session by $\mathbf{c} \downarrow f_0$. The end is there to ensure all processes in the session terminate. The valid use of futures is examined during generating object types from a global type, a procedure introduced in Section 4.3. If $\mathbf{s} \uparrow f''$ is specified before $\mathbf{d}{\downarrow} f''$, the projection procedure will return *undefined* since f'' can not be read before being resolved.

4.2 Local Types

Besides global types, to statically check code, we define local types, which describe local specifications at object level. The syntax of local types is defined:

Definition 2.

$$\begin{aligned} \mathbf{L} \quad ::= \; & \mathbf{p}!_f m.\mathbf{L} \mid \mathbf{p}?_f m.\mathbf{L} \mid \mathsf{Put}\ f : (\mathtt{C}).\mathbf{L} \mid \mathsf{Get}\ f : (\mathtt{C}).\mathbf{L} \mid \mathsf{Await}(f, f').\mathbf{L} \\ & \mid \mathsf{React}(f).\mathbf{L} \mid \oplus\{\mathbf{L}_j\}_{j\in J} \mid \&_f\{\mathbf{L}_j\}_{j\in J} \mid \mathbf{L}^*.\mathbf{L} \mid \mathsf{skip}.\mathbf{L} \mid \mathsf{end} \end{aligned}$$

We use . to denote sequential composition. The type $\mathbf{p}!_f m$ denotes a sending action via an asynchronous remote call on method m at endpoint **p**. The type $\mathbf{p}?_f m$ denotes a receiving action which starts a new process computing f by executing method m after a call from **p**. The resolving $\mathsf{Put}\ f : (\mathtt{C})$ and fetching $\mathsf{Get}\ f : (\mathtt{C})$ have the same intuitive meaning as their global counterparts. The suspension $\mathsf{Await}(f, f')$ means that the process computing f suspends its action until future f' is resolved. The reactivation $\mathsf{React}(f)$ means the process continues the execution with f. The choice operator $\oplus$ in $\oplus\{\mathbf{L}_j\}_{j\in J}$ denotes that the currently active process selects a branch to continue. The offer operator $\&_f$ in $\&_f\{\mathbf{L}_j\}_{j\in J}$ denotes that the object offers branches $\{\mathbf{L}_j\}_{j\in J}$ when f is resolved. The type skip denotes no action and we say $\mathbf{L}.\mathsf{skip} \equiv \mathbf{L} \equiv \mathsf{skip}.\mathbf{L}$.

In ABS-ST the communication happens among processes in different objects. We list three kinds of local types:

- *A method type* describes the execution of a single process on a particular future f. It has the following attributes: (1) Its first action is $\mathbf{p}?_f m$ for some $\mathbf{p}$, m, f, and (2) if it has a branching type, the final action in every branch is $\mathsf{Put}\ f:(\mathtt{C})$ for some $\mathtt{C}, f$, and (3) it contains no further resolving action or receiving action, and (4) it contains no end.
- *An object type* is a type which is not a method type.
- *A condensed type*, denoted by $\hat{\mathbf{L}}$, where $\mathbf{L}$ is an object type, replaces every action, except receiving and reactivation actions, in $\mathbf{L}$ with skip.

Example 3. Consider object $\mathbf{d}$ in the *grading system* in Section 2. Its method type on future f, which is used for calling method *publish*, is $\mathbf{c}?_f \mathit{publish}.\mathsf{Put}\ f$. Its object type is $\mathbf{L} = \mathbf{c}?_f \mathit{publish}.\mathsf{Put}\ f.\mathbf{s}?_{f''} \mathit{request}.\mathsf{Put}\ f''.\mathsf{end}$, and its condensed type is $\hat{\mathbf{L}} = \mathbf{c}?_f \mathit{publish}.\mathsf{skip}.\mathbf{s}?_{f''} \mathit{request}.\mathsf{skip}.\mathsf{skip} \equiv \mathbf{c}?_f \mathit{publish}.\mathbf{s}?_{f''} \mathit{request}$.

4.3 Projection

Projection is the procedure to derive local types of endpoints from a global type. Since in SABS data is sent between different objects by active processes, the projection rules have two levels: (1) Projecting a global type on objects and resulting *object types* and (2) projecting *object types* on a *future* and resulting *method types*, which type the behavior of process for computing the target future.

4.4 Projecting a Global Type to Local Types

We say a global type is *projectable* if every projection on every of its participants is defined and every future is introduced exactly once (i.e. linearity). A projectable global type implies that the futures appear in it are located correctly across multiple objects; thus the object types gained from it ensure the correct usage of futures.

We define $\mathit{pre}(\mathbf{G}, \mathbf{G}')$ as the set of prefixes of $\mathbf{G}$:

$$\mathit{pre}(\mathbf{G}, \mathbf{G}') = \{\mathbf{G}'' \mid \mathbf{G} \in \mathbf{G}' \text{ implies } \mathbf{G}''.\mathbf{G} \in \mathbf{G}'\}$$

and that a future f *introduced in* $\mathbf{G}'$ is *active* on object $\mathbf{o}$ in $\mathbf{G}$ iff (if and only if):

$$\begin{aligned}&(\mathbf{p} \xrightarrow{f} \mathbf{o} \in \mathit{pre}(\mathbf{G}, \mathbf{G}')) \wedge (\mathbf{o}{\downarrow} f:(\mathtt{C}) \notin \mathit{pre}(\mathbf{G}, \mathbf{G}'))\\ \wedge &\left(\left(\mathsf{Rel}(\mathbf{p}, f') \in \mathit{pre}(\mathbf{G}, \mathbf{G}') \wedge\ f \text{ active in } \mathsf{Rel}(\mathbf{p}, f')\right) \to \mathbf{o}{\downarrow} f' \in \mathit{pre}(\mathbf{G}, \mathbf{G}')\right)\end{aligned}$$

The first conjunct captures that after $\mathbf{p} \xrightarrow{f} \mathbf{o}$, f becomes active on $\mathbf{o}$, while after $\mathbf{o}{\downarrow} f:(\mathtt{C})$, f becomes inactive on $\mathbf{o}$; the second conjunct captures that if f has been suspended on f' (i.e. $\mathsf{Rel}(\mathbf{p}, f') \in \mathit{pre}(\mathbf{G}, \mathbf{G}') \wedge f$ active in $\mathsf{Rel}(\mathbf{p}, f')$), then f must have been reactivated by resolving f' (i.e. $\mathbf{o}{\downarrow} f' \in \mathit{pre}(\mathbf{G}, \mathbf{G}')$).

Projection from global types to object types

$$pj(\mathbf{p} \xrightarrow{f} \mathbf{q}{:}m, \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathbf{q}!_f m & \text{if } (\mathbf{o} = \mathbf{p}) \wedge (f \text{ is fresh}) \\ \mathbf{p}?_f m & \text{if } (\mathbf{o} = \mathbf{q}) \wedge (f \text{ is fresh}) \wedge (\mathbf{q} \text{ is inactive}) \\ \mathsf{skip} & \text{if } (\mathbf{o} \neq \mathbf{q} \wedge \mathbf{o} \neq \mathbf{p}) \wedge (f \text{ is fresh}) \end{cases}$$

$$pj(\mathbf{p}{\downarrow}f{:}(\mathtt{C}), \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathsf{Put}\ f{:}(\mathtt{C}) & \text{if } (\mathbf{o} = \mathbf{p}) \wedge (f \text{ is active on } \mathbf{p}) \\ \mathsf{React}\ f' & \begin{array}{l}\text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (\mathsf{Rel}(\mathbf{o}, f) \in \mathit{pre}(\mathbf{p}{\downarrow}f{:}(\mathtt{C}), \mathbf{G})) \\ \qquad \wedge (\ f' \text{ is active on } \mathbf{o} \text{ in } \mathsf{Rel}(\mathbf{o}, f))\end{array} \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

$$pj(\mathbf{p}{\uparrow}f{:}(\mathtt{C}), \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathsf{Get}\ f{:}(\mathtt{C}) & \text{if } (\mathbf{o} = \mathbf{p}) \wedge (\mathbf{q}{\downarrow}f{:}(\mathtt{C}) \in \mathit{pre}(\mathbf{p}{\uparrow}f{:}(\mathtt{C}), \mathbf{G})) \\ \mathsf{skip} & \text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (\mathbf{q}{\downarrow}f{:}(\mathtt{C}) \in \mathit{pre}(\mathbf{p}{\uparrow}f{:}(\mathtt{C}), \mathbf{G})) \end{cases}$$

$$pj(\mathsf{Rel}(\mathbf{p}, f), \mathbf{o})_{\mathbf{G}} = \begin{cases} \mathsf{Await}(f', f) & \begin{array}{l}\text{if } (\mathbf{o} = \mathbf{p}) \wedge (f' \text{ is active on } \mathbf{p}) \\ \qquad \wedge (\nexists \mathbf{p}'.\ \ \mathbf{p}'{\downarrow}f \in \mathit{pre}(\mathsf{Rel}(\mathbf{p}, f), \mathbf{G}))\end{array} \\ \mathsf{skip} & \text{if} (\mathbf{o} \neq \mathbf{p}) \end{cases}$$

$$pj(\mathbf{p}\{\mathbf{g}_j\}_{j \in J}, \mathbf{o})_{\mathbf{G}} = \begin{cases} \oplus\{pj(\mathbf{g}_j, \mathbf{o})_{\mathbf{G}}\}_{j \in J} & \text{if } (\mathbf{o} = \mathbf{p}) \\ \&_f\{pj(\mathbf{g}_j, \mathbf{o})_{\mathbf{G}}\}_{j \in J} & \text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (f \text{ is active on } \mathbf{o}) \\ \mathbf{L} & \text{if } (\mathbf{o} \neq \mathbf{p}) \wedge (\forall j \in J.\ pj(\mathbf{g}_j, \mathbf{o})_{\mathbf{G}} = \mathbf{L}) \end{cases}$$

$$pj(\mathsf{end}, \mathbf{o})_{\mathbf{G}} = \mathsf{end} \qquad pj(\mathbf{g}^*, \mathbf{o})_{\mathbf{G}} = (pj(\mathbf{g}, \mathbf{o})_{\mathbf{G}})^* \text{ if } \mathbf{g} \text{ is self-contained}$$

$$pj((\mathbf{G}'.\mathbf{g}), \mathbf{o})_{\mathbf{G}} = \begin{cases} pj(\mathbf{G}', \mathbf{o})_{\mathbf{G}} & \text{if } (pj(\mathbf{G}', \mathbf{o})_{\mathbf{G}} = \mathsf{skip}) \\ pj(\mathbf{G}', \mathbf{o})_{\mathbf{G}}.pj(\mathbf{g}, \mathbf{o})_{\mathbf{G}} & \text{otherwise} \end{cases}$$

Projection from object types to method types

$$\mathsf{SIMPLE} = \{\mathbf{p}!_{f'} m, \mathbf{p}?_f m, \mathsf{Put}\ f{:}(\mathtt{C}), \mathsf{Get}\ f'{:}(\mathtt{C}), \mathsf{Await}(f, f')\}$$

$$pjm(\mathbf{L}, f) = \begin{cases} \mathbf{L} & \text{if } (\mathbf{L} \in \mathsf{SIMPLE}) \wedge (f \text{ is active}) \\ \mathsf{skip} & \text{if } (\mathbf{L} \in \mathsf{SIMPLE}) \wedge (\text{some other future is active}) \\ \mathsf{skip} & \text{if } (\mathbf{L} = \mathsf{React}(f)) \vee (\mathbf{L} = \mathsf{end}) \\ pjm(\mathbf{L}_1, f).pjm(\mathbf{L}_2, f) & \text{if } (\mathbf{L} = \mathbf{L}_1.\mathbf{L}_2) \\ pjm(\mathbf{L}', f) & \text{if } (\mathbf{L} = (\mathbf{L}')^*) \wedge (\mathbf{p} \xrightarrow{f} \mathbf{q} \in \mathbf{L}') \\ pjm(\mathbf{L}', f)^* & \text{if } (\mathbf{L} = (\mathbf{L}')^*) \wedge (\mathbf{p} \xrightarrow{f} \mathbf{q} \notin \mathbf{L}') \\ pjm(\mathbf{L}_j, f) & \begin{array}{l}\text{if } ((\mathbf{L} = \&_{f'}\{\mathbf{L}_j\}_{j \in J}) \vee (\mathbf{L} = \oplus\{\mathbf{L}_j\}_{j \in J})) \\ \quad \wedge (f \neq f') \wedge (\mathbf{p} \xrightarrow{f} \mathbf{o} \in \mathbf{L}_j)\end{array} \\ \oplus\{pjm(\mathbf{L}_j, f)\}_{j \in J} & \text{if } (\mathbf{L} = \oplus\{\mathbf{L}_j\}_{j \in J}) \\ \&_{f'}\{pjm(\mathbf{L}_j, f)\}_{j \in J} & \text{if } (\mathbf{L} = \&_{f'}\{\mathbf{L}_j\}_{j \in J}) \wedge (\mathbf{L}_j \text{ are distinct}) \end{cases}$$

Fig. 4. Projection rules

Fig. 4 defines the projection rules as a function $pj(\mathbf{g}, \mathbf{o})_{\mathbf{G}}$ projecting $\mathbf{g}$ to object $\mathbf{o}$, where $\mathbf{g} \in \mathbf{G}$. We write $pj(\mathbf{G}, \mathbf{o})_{\mathbf{G}} = \mathbf{G} \upharpoonright \mathbf{o}$. The side-conditions verify the defined cases, where the futures are used correctly; others are undefined.

The interaction type projects a sending action on the caller side and a receiving action on the callee side. A resolving type gives an action for resolving f on the corresponding object, and generates a reactivation for every objects who are waiting for f; for others, it gives skip. A fetching type gives an action for fetching the result from f on the corresponding object and gives skip for others. Its side-condition ensures that a future is resolved before fetching it. This must be checked at a global level because resolving and fetching take place in different objects. A releasing type gives suspension for the corresponding object and gives skip for others. Its side-condition ensures that the releasing object does not have any other future waiting for the same resolving. A branching type gives a choice type for the active side and an offer for every one that either receives one of the calls invoked by the object making the choice or reads from the active future. For other objects, each branch should have the same behavior so that those objects always know how to proceed no matter which branch was selected. Termination type gives end, which means every future has been resolved and all objects are inactive. The repetition and concatenation are propagated down.

4.5 Projecting Object Types to Method Types

Fig. 4 also defines the projection of an object type $\mathbf{L}$ to a method type on a future f, denoted by $pjm(\mathbf{L}, f)$. Since the correct usage of futures has been checked when we do $\mathbf{G} \upharpoonright \mathbf{o}$, $pjm(\mathbf{G} \upharpoonright \mathbf{o}, f)$ has ensured the valid usage of futures.

For a sending, receiving, resolving, fetching, suspending, or repetition object type which is active on future f, the projected method type is itself; otherwise the projection gives skip. A choice object type gives a choice method type when it projects on an active future used by the target object, while gives a unique $\mathbf{L}$ when it projects on a future which only appears in one branch. Similarly for the case of offer object type. A reactivation object type gives a method type skip when it projects on any future because the next action after a suspension will always be a reactivation inside a method type. Termination object type also always gives skip for any future because it is not visible. A concatenation object type gives a concatenation method type on any future.

5 Type System

We say the objects involving in a sequence of communications, i.e. a session, satisfy *communication correctness* iff, during the interactions, they always comply with some pre-defined global type. To locally check endpoint implementations and statically ensure communication safety, which is currently not supported by core ABS, we here introduce a type system, which is defined in Fig. 5.

We use Θ as session environments, which contain sessions associating to global types that they follow, with information about the types of participants; we use Γ as shared environments mapping expressions to ground types, and Δ as channel environments mapping channels (composed by a session name and an object) to local types. Note that, channel environments *only exist in the type system*. When we write Θ, Θ', we mean $domain(\Theta) \cap domain(\Theta') = \emptyset$, so as for φ, Γ and Δ. Θ and Γ together store the shared information. For convenience, we define $role(\mathbf{G})$ returning the set of participants in $\mathbf{G}$, $ptypes(C)$ returning the

$$\begin{array}{rl} \Theta(\text{Session Environments}) ::= \emptyset \mid \Theta, \{\varphi\}_{\mathbf{s}:\mathbf{G}} & \quad \varphi ::= \emptyset \mid \varphi, I : \mathbf{p} \mid \varphi, \mathsf{Any} : \mathbf{0} \\ \Gamma(\text{Shared Environments}) ::= \emptyset \mid \Gamma, e : T & \\ \Delta(\text{Channel Environments}) ::= \emptyset \mid \Delta, \mathbf{s}[\mathbf{p}] : \mathbf{L} & \end{array}$$

(T-New-Session)

$$\frac{\mathbf{G}\text{ is projectable }\ \ \mathbf{s}\text{ fresh }\ \ \Theta = \{\mathsf{Any}:\mathbf{0}\}_{\mathbf{s}:\mathbf{G}}\ \ \ \Gamma\vdash e:\mathsf{String}}{\Theta,\Gamma\vdash [\texttt{Protocol: G }]\ \mathsf{Ses}\ \mathbf{s} = \mathbf{new}\ \ \mathit{Session}(\mathrm{e}) \triangleright \Delta, \mathbf{s}[\mathbf{0}]:\mathbf{G}\upharpoonright\mathbf{0}}$$

(T-New-Join)

$$\frac{\Theta \vdash \{\varphi\}_{\mathbf{s}:\mathbf{G}}\text{ for some }\mathbf{G}\ \ \ \Gamma\vdash\overline{e}:\mathit{ptypes}(C)\ \ \ \Gamma\vdash c:I}{\Theta,\Gamma\vdash[\texttt{Ses: s}]\ \ c = \mathbf{new}\ C[(\overline{e})]\triangleright\Delta,\mathbf{s}[c]:\emptyset}$$

(T-Scheduler)

$$\frac{\begin{array}{c}\forall I\in\overline{I}.\mathit{implements}(C,I)\ \ \ \Theta,\Gamma[\texttt{this}\mapsto C,\mathit{fields}(C)]\vdash\overline{M}\triangleright\Delta,\mathbf{s}[\mathit{obj}(C)]:\emptyset \\ \Theta=\Theta'',\{\varphi\}_{\mathbf{s}:\mathbf{G}}\ \ \ \exists\mathbf{p}\in\mathit{role}(\mathbf{G})\ s.t.\ \mathbf{G}\upharpoonright\mathbf{p}=\mathbf{L}\ \ \ \Theta'=\Theta'',\{\varphi,I:\mathbf{p}\}_{\mathbf{s}:\mathbf{G}}\end{array}}{\Theta',\Gamma\vdash[\texttt{Scheduler: L }]\,\texttt{class}\,C\,[(\overline{T}\ \overline{x})]\,[\texttt{implements}\,\overline{I}]\,\{\,\overline{T}\ \overline{x};\ \overline{M}\}\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}}$$

(T-Send)

$$\frac{\Theta=\Theta',\{\varphi,I:\mathbf{q}\}_{\mathbf{s}:\mathbf{G}}\ \ \ \Gamma\vdash e:I\ \ \ \Gamma\vdash x:f\ \ \ \Gamma\vdash s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}}{\Theta,\Gamma\vdash x=e!m(\overline{e});s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{q}!_f m.\mathbf{L}}$$

(T-Method)

$$\frac{\begin{array}{c}(\Theta=\Theta',\{\varphi,T:\mathbf{q}\}_{\mathbf{s}:\mathbf{G}})\wedge((T=I)\vee(T=\mathsf{Any})) \\ T\in\overline{T}\ \ \ \Gamma'=\Gamma[\overline{x}\mapsto\overline{T},\overline{x'}\mapsto\overline{T'}]\ \ \ \Gamma'[\texttt{destiny}\mapsto f']\vdash s\triangleright\Delta,\mathit{pjm}(\mathbf{L},f')\end{array}}{\Theta,\Gamma\vdash T''\ m\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};s\}\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathit{pjm}(\mathbf{q}?_f m.\mathbf{L},f')}$$

(T-Offer)

$$\frac{\Gamma\vdash e:T\ \ \ J=\{1..n\}\ \ \ \forall j\in J.\Gamma\vdash p_j:T\ \ \ \Gamma\vdash s_j\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}_j\ \ \ f\text{ active on }\mathbf{p}\text{ in }\mathbf{L}_j}{\Gamma\vdash\texttt{case}\ e\{p_1\Rightarrow s_1,...,p_n\Rightarrow s_n\}\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\&_f\{\mathbf{L}_j\}_{j\in J}}$$

(T-Choice)

$$\frac{\Gamma\vdash e:T\ \ \ \Gamma\vdash p:T\ \ \ J=\{1..n\}\ \ \ k\in J\ \ \ \Gamma\vdash s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}_k}{\Gamma\vdash\texttt{case}\ e\ \{p\Rightarrow s\}\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\oplus\{\mathbf{L}_j\}_{j\in J}}$$

(T-Await)

$$\frac{\mathbf{L}=\mathsf{React}(f).\mathbf{L}'\ \ \ f'\text{ inactive on }\mathbf{p}\text{ in }\mathbf{L}'\ \ \ \Gamma\vdash s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}}{\Gamma\vdash\texttt{await}\ e?;s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathsf{Await}(f,f').\mathbf{L}}$$

(T-Skip)

$$\frac{\Gamma\vdash s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}}{\Gamma\vdash\texttt{skip};s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathsf{skip}.\mathbf{L}}$$

(T-Return)

$$\frac{\Gamma\vdash e:T\ \ \ \Gamma(\texttt{destiny})=f\ \ \ \Gamma\vdash s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}}{\Gamma\vdash\texttt{return}\ e;s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathsf{Put}\ f:(\texttt{C}).\mathbf{L}}$$

(T-While)

$$\frac{\Gamma\vdash b:\mathsf{Bool}\ \ \ \Gamma\vdash s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}}{\Gamma\vdash\texttt{while}\ b\{s\}\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}^\star}$$

(T-Get)

$$\frac{\Gamma\vdash s\{e.\mathbf{get}/x\}\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathbf{L}}{\Gamma\vdash x=e.\mathbf{get};s\triangleright\Delta,\mathbf{s}[\mathbf{p}]:\mathsf{Get}\ f:(\texttt{C}).\mathbf{L}}$$

Fig. 5. The type system for the concurrent object level of ABS (Parts: Session-related)

types of parameters of C, $\mathit{implements}(C, I)$ returning true if C can implement interface I, $\mathit{obj}(C)$ returning an instance of class C, and $\mathit{fields}(C)$ returning a shared environment containing attributes of C. We only list the session-related typing rules related. Others are as same as those in core ABS [15].

Rule **(T-New-Session)** types a session creation by checking if $\mathbf{G}$ in the annotation is projectable (see Section 4.4), the session id s is fresh, and the type of e is String. If all conditions are satisfied, we create $\Theta = \{\mathsf{Any} : \mathbf{0}\}_{\mathsf{s}:\mathbf{G}}$ to record mappings of types to the participants in $\mathbf{G}$. The first mapping is $\mathsf{Any} : \mathbf{0}$, in which Any types the session initializer. Also, a channel $\mathsf{s}[\mathbf{0}]$ and its type $\mathbf{G} \upharpoonright \mathbf{0}$ is created in shared environments to specify this object playing $\mathbf{0}$ in $\mathbf{G}$. Rule **(T-New-Join)** types an object creation, which joins session s, which is a name (with type String) of a session. The object creation is valid if s has been created (i.e. $\Theta \vdash \{\varphi\}_{\mathsf{s}:\mathbf{G}}$) and the type of $\overline{e}$ is $\mathit{ptypes}(C)$.

Rule **(T-Scheduler)** is the key rule to activate session-based typing. A class with annotation **[Scheduler: L]** is well-typed if its methods are well-typed (this part is as same as the rule **(T-Class)** in [15]) and, by given the fact that the instance of C has joined session s (i.e. $\mathsf{s}[\mathit{obj}(C)] : \emptyset$), the local scheduler who specifies the behavior of $\mathit{obj}(C)$ against $\mathbf{L}$ should find $\mathbf{L} = \mathbf{G} \upharpoonright \mathbf{p}$ where $\mathbf{p} \in \mathit{role}(\mathbf{G})$, which implies that $\mathit{obj}(C)$, typed by I, plays as $\mathbf{p}$ in $\mathbf{G}$ when it joins s. Then we extend Θ to Θ' by adding $I : \mathbf{p}$ into $\{\varphi\}_{\mathsf{s}:\mathbf{G}}$ to claim that $\mathbf{p}$ associates to interface I, and replacing $\mathsf{s}[\mathit{obj}(C)] : \emptyset$ with $\mathsf{s}[\mathbf{p}] : \mathbf{L}$ in the channel environment.

Rule **(T-Send)** types an asynchronous remote method call. The object is allowed to have such a call, specified by $\mathsf{s}[\mathbf{p}] : \mathbf{q}!_f m$, when the object calls a method m using f (by checking $x : f$) at an object playing $\mathbf{q}$ in $\mathbf{G}$ (i.e. Θ has $\{\varphi, I : \mathbf{q}\}_{\mathsf{s}:\mathbf{G}}$ and Γ has $e : I$) and its next statement s is also well-typed.

Rule **(T-Method)** types a method execution. It is valid to do so if the method body is well-typed and the caller is an object playing $\mathbf{q}$ in $\mathbf{G}$, known by $\Theta = \Theta', \{\varphi, T : \mathbf{q}\}_{\mathsf{s}:\mathbf{G}}$ where T is either equal to I or Any. We use f' for returning the computation result as long as $f' = f$ (i.e. $\mathit{pjm}(\mathbf{q}?_f m.\mathbf{L}, f') = \mathbf{q}?_f m.\mathit{pjm}(\mathbf{L}, f')$.)

Rule **(T-Offer)** types **case** $e\{p_1 \Rightarrow s_1, ..., p_n \Rightarrow s_n\}$ with $\&_f\{\mathbf{L}_j\}_{j\in J}$ by checking if every branch $p_j \Rightarrow s_j$ is well-typed by $\mathbf{L}_j$ and checking if f is active in $\mathbf{L}_j$ on the object $\mathbf{p}$ in session s. Rule **(T-Choice)** is the counterpart of **(T-Offer)**. Rule **(T-Await)** types the await statement with $\mathsf{Await}(f, f')$. It checks if the next statament is well-typed by $\mathsf{React}(f).\mathbf{L}'$, which specifies the next action is to reactivate the usage of f and, since f' has been resolved, in $\mathbf{L}'$ we have f' inactive on $\mathbf{p}$. Other rules are straightforward.

After locally type checking every objects' implementations in a session based on their corresponding local types, which are projected from a global type that the session follows, our system ensures overall interactions among those objects comply with communication correctness:

Theorem 1 (Communication Correctness). *Let* $\mathbf{G}$ *be a projectable global type and* S *a closed system in which a session* s *obeys to* $\mathbf{G}$*. Let* $\mathbf{p}'_1, ..., \mathbf{p}'_n$ *are objects in* s *and respectively act as* $\mathbf{p}_1, ..., \mathbf{p}_n$ *in* $\mathbf{G}$*. If the objects' implementations are all well-typed by rules in Fig. 5, the interactions among* $\mathbf{p}'_1, ..., \mathbf{p}'_n$ *comply with communication correctness against* $\mathbf{G}$*.*

6 Session Automata

As we type check a SABS program via rules in Fig.5, by rule **(T-Scheduler)**, a local type $\mathbf{L}$ is assigned as a *scheduling policy* to the scheduler of object

[`Scheduler`: **L**] **`class`** $C\{...\}$; the scheduler can (re)activate processes (i.e. by executing methods) based on **L**. To ensure that the scheduler's behavior follows **L**, we propose a verification mechanism where a scheduler uses a session automaton [2], as a *scheduling policy*, to model the possible sequence of events.

In this model, when the object is idle, the object's scheduler inputs the processes which can be (re)activated according to the session automaton, which is automatically generated by **L**. If a labelled transition, which corresponds to an event, can fire in the automaton, the object (re)activates this event. If there are several processes which can run such transition, the scheduler randomly selects one of them. This mechanism is a variant of typestate [20].

Session automata, a subclass of register automata [17], only store fresh futures; it is decideable whether two session automata accept the same language [2]:

Definition 3. [k-Register Session Automata] Let Σ be a finite set of labels, D be an infinite set of data equipped with equality, and $k \in \mathbb{N}$. A *k-Register Session Automaton* is a tuple (Q, q_0, Φ, F) where Q is the finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\Phi \subseteq (Q \times Q') \cup (Q \times \Sigma \times 2^{\{1,\dots,k\}} \times \{1, \dots, k\} \times Q)$ is the transition relation.

Data words are words over an alphabet $\Sigma \times D$. A data word automata has a data store, which can save k data values. A transition fires for a letter $(a, d) \in \Sigma \times D$ if a set of equalities of the form $d = r_i$ are satisfied, where r_i refers to the ith stored data value. After a transition fires, the data store records d.

Let $\sigma : \{1, \dots, k\} \to D$ be the store. We define (q, a, I, i, q') as a transition in automaton from state q to state q' upon reading (a, d) if $\sigma[i] = d$ for all $i \in I$, $I = \{1..n\}$ by updating $\sigma[i]$ to d, and define (q, q') as an ϵ-transition that switches the state without reading the next letter:

Definition 4. [Runs of Session Automata] A *run* of a k-register session automaton $A = (Q, q_0, \Phi, F)$ on a data word $w = (a_1, d_1), \dots, (a_k, d_k) \in (\Sigma \times D)^k$ is a sequence $s \in \big(Q \times \mathbb{N} \times (\{1, \dots, k\} \to D)\big)^*$. An element (q, j, σ) of the sequence denotes that A is at state q with store σ and reads (a_j, d_j). To be a run of A, the sequence $s = (q_0, j_0, \sigma_0), \dots, (q_n, j_n, \sigma_n)$ must satisfy the following:

$$\Big((q_i, q_{i+1}) \in \Phi \land (j_i = j_{i+1}) \land (\sigma_i = \sigma_{i+1})\Big) \lor$$
$$\Big((q_i, (a_{j_i}, d_{j_i}), I, k, q_{i+1}) \in \Phi \land (j_{i+1} = j_i + 1) \land (\sigma_{i+1} = \sigma_i[k/d_{j_i}]) \land \forall l \in I.\ \sigma_i(l) = d_{j_i}\Big)$$

where $I = \{1..n\}$ and $\sigma_i[k/d_{j_i}]$ is a function mapping the kth stored data to d_{j_i}. Now we revise Σ to $\Sigma = \big((\{\texttt{invocREv}\} \times \mathsf{Met}) \cup \{\texttt{reactEv}\}\big)$ and $D = \mathsf{Fut}$, where `invocREv` labels process activation and `reactEv` labels process reactivation. Given an object type, we can build a session automaton:

Definition 5. Let **L** be an object type. Let k be number of futures in $\hat{\mathbf{L}}$. We assume the futures are ordered and $pos(f)$ refers to the number of f in the ordering. The k-register session automaton $A_{\mathbf{L}}$ is defined inductively as follows:

– $\mathbf{p}?_f m$ is mapped to a 2-state automaton which reads $(\texttt{invocREv}, m)$ and stores the future f in the $pos(f)$-th register on its sole transition.

– React f is mapped to a 2-state automaton which reads `reactEV` and tests for equality with the $pos(f)$-th register on its sole transition.
– Concatenation, branching, and repetition using the standard construction for concatenation, union, and repetition for NFAs.

When a process is activated, the automaton stores the process's corresponding futures; when a process is reactivated, the automaton compares the process's corresponding futures with the specified register. As all repetitions in types projected from a global type are self-contained, after the repetition, the futures used there are resolved and thus the automaton can overwrite it safely. The example below shows how a session automaton works based on an object type:

Example 4. Consider the example from Section 2. A simple automaton describing the sequence for the **d** (Service Desk) is

$$\begin{tikzcd}
\text{start} \arrow[r] & 1 \arrow[r, "(\texttt{invocREv}, \mathsf{publish})", "d \mapsto r_0"'] & 2 \arrow[r, "(\texttt{invocREv}, \mathsf{request})", "d \mapsto r_1"'] & 3
\end{tikzcd}$$

The scheduler above does not need to use registers because it does not have reactivations. The following one must read the registers to schedule reactivations:

$$\mathbf{L} = (\mathbf{p}?_f m_1.\mathsf{Await}(f, f').\mathbf{p}?_{f''} m_3.\mathsf{Put}\ f''.\mathsf{React}(f).\mathsf{Put}\ f)^*$$

The generated 2-register session automaton is:

$$\begin{tikzcd}
\text{start} \arrow[r] & 1 \arrow[r, "(\texttt{invocREv}, m_1)", "d \mapsto r_0"'] & 2 \arrow[r, "(\texttt{invocREv}, m_3)", "d \mapsto r_1"'] & 3 \arrow[r, "\texttt{reactEv}", "d = r_0"'] & 4 \arrow[ll, bend left, "(\texttt{invocREv}, m_1)", "d \mapsto r_0"']
\end{tikzcd}$$

The following theorem states that the objects involving in a session are faithful to the session's protocol if their processes can be verified by the corresponding schedulers, whose behaviors follow the session automata:

Theorem 2 (Fidelity). *Let $\boldsymbol{G}$ be a projectable global type and S a closed system in which a session $\boldsymbol{s}$ obeys to $\boldsymbol{G}$. Let $\mathbf{p'_1}, ..., \mathbf{p'_n}$ are objects interacting in $\boldsymbol{s}$ and respectively act as participants $\mathbf{p_1}, ..., \mathbf{p_n}$ in $\boldsymbol{G}$. Let A_j be a session automaton generated from $\boldsymbol{G} \upharpoonright \mathbf{p_j}$. If every scheduler for $\mathbf{p_j}$, $j \in \{1..n\}$ accepts the same language as A_j does, the implementations on objects are faithful to $\boldsymbol{G}$.*

7 Related and Future Work

The compositional approach introduced in [5, 6] proposed a four event semantics for core ABS. Their verification approach was bottom-up, i.e., class invariants are verified and composed into system property based on history well-formedness; while our approach is top-down, i.e., system property is specified in session types and projected into class invariants. We verify class invariants based on the scheduling policy type-checked by local session types. Besides, we introduce session types for process suspension and process reactivation.

Session types for object-oriented languages have been studied in [3, 8] and implemented for libraries/extensions of mainstream languages like Java [13].

Also, lightweight session programming in Scala [19] was proposed by introducing a representation of session types as Scala types. However, they do not explore the valid usage of futures for modeling channel-based concurrency, neither verify cooperative scheduling against specified execution orders.

Schedulers with automata for actor-based models were studied by Jaghoori et al. [14], while user-defined schedulers for ABS were introduced by Bjørk et al. [1]. Our use of automata is similar to the *drivers* of [14], where drivers can not reject any process if the process queue is non-empty and do not consider reactivations. Our schedulers enable the object to wait for a method call to arrive.

Field et al. [7] used finite state automata (without registers) to encode typestate, Gay et al. [8] used typestate to guide session types with non-uniform objects, while Grigore et al. [9] established register automata for runtime verification. In their approach the automata monitor the order of method invocations in a sequential setting. The registers are used to store an unbounded amount of object identities. Our automata are extended to be able to check the specified orders of method calls. The schedulers thus can apply these automta to schedule specified activations and reactivations in a concurrent setting.

Neykova and Yoshida [18] also consider an actor model with channels, where processes are monitored by automata. Deniélou and Yoshida [4] used communicating automata to approximate processes and local types. However, their approaches do not consider scheduling and validating the usage of futures.

We plan to prove that our type system ensures that interactions among objects are deadlock-free and always progresses, and then implement a session-based extension for the core ABS language.

8 Conclusion

We establish a hybrid framework for compositional analysis. The system property is guaranteed by type checking each objects' behaviors against local session types, which are gained by projecting global types on endpoints. In summary, we statically ensure communication correctness for concurrent processing and, at the same time, ensure local schedulers' behaviors will follow the specified execution order among asynchronous communications at runtime.

9 Acknowledgments

We thank Reiner Hähnle and Patrick Eugster who provided the original idea and insightful discussions for this paper. We also thank the reviewers for their constructive comments. This work was supported by the ERC grant FP7-617805 *LiVeSoft: Lightweight Verification of Software* and the EU project FP7-610582 *Envisage: Engineering Virtualized Services.*

References

1. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. User-defined schedulers for real-time concurrent objects. *ISSE*, 9(1):29–43, 2013.
2. B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In M. Béal and O. Carton, editors, *DLT'13*, volume 7907 of *LNCS*, pages 118–130. Springer, 2013.

3. J. Campos and V. T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In K. Honda and A. Mycroft, editors, *PLACES'10*, volume 69 of *EPTCS*, pages 12–28, 2010.
4. P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In H. Seidl, editor, *ESOP'12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
5. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In A. P. Felty and A. Middeldorp, editors, *CADE'15*, volume 9195 of *LNCS*, pages 517–526. Springer, 2015.
6. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
7. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Sci. Comput. Program.*, 58(1-2):57–82, 2005.
8. S. J. Gay, N. Gesbert, A. Ravara, and V. T. Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.
9. R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In N. Piterman and S. A. Smolka, editors, *TACAS'13*, volume 7795 of *LNCS*, pages 260–276. Springer, 2013.
10. R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, editors, *FMCO'12*, volume 7866 of *LNCS*, pages 1–37. Springer, 2012.
11. R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, 1985.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *POPL'08*, pages 273–284. ACM, 2008.
13. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In J. Vitek, editor, *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
14. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *The Journal of Logic and Algebraic Programming*, 78(5):402 – 416, 2009.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2010.
16. E. Kamburjan. Session Types for ABS. Technical report, 2016. www.se.tu-darmstadt.de/publications/details/?tx_bibtex_pi1[pub_id]=tud-cs-2016-0179.
17. M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
18. R. Neykova and N. Yoshida. Multiparty session actors. In E. Kühn and R. Pugliese, editors, *COORDINATION'14 Proceedings*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.
19. A. Scalas and N. Yoshida. Lightweight session programming in Scala. In S. Krishnamurthi and B. S. Lerner, editors, *ECOOP'16*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
20. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.
21. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.