Session Types for Link Failures

Manuel Adameit, Kirstin Peters, Uwe Nestmann

Session Types for Link Failures

Manuel Adameit, Kirstin Peters, and Uwe Nestmann

TU Berlin, Germany

Abstract. We strive to use session type technology to prove behavioural properties of fault-tolerant distributed algorithms. Session types are designed to abstractly capture the structure of (even multi-party) communication protocols. The goal of session types is the analysis and verification of the protocols' behavioural properties. One important such property is progress, i.e., the absence of (unintended) deadlock. Distributed algorithms often resemble (compositions of) multi-party communication protocols. In contrast to protocols that are typically studied with session types, they are often designed to cope with system failures. An essential behavioural property is (successful) termination, despite failures, but it is often elaborate to prove for distributed algorithms.
We extend multi-party session types with optional blocks that cover a limited class of link failures. This allows us to automatically derive termination of distributed algorithms that come within these limits.

1 Introduction

Session types are used to statically ensure correctly coordinated behaviour in systems without global control. One important such property is progress, i.e., the absence of (unintended) deadlock. Like with every other static typing approach, the main advantage is that the respective properties are then provable without unrolling the process, i.e., without computing its executions. Thereby, the state explosion problem is avoided. Hence, after the often elaborate task of establishing a type system, they allow to prove properties of processes in a quite efficient way.

Session types describe global behaviours as *sessions*, i.e., units of conversations. The participants of such sessions are called *roles*. *Global types* specify protocols from a global point of view, whereas *local types* describe the behaviour of individual roles within a protocol. *Projection* ensures that a global type and its local types are consistent. These types are used to reason about processes formulated in a *session calculus*. Most of the existing session calculi are extensions of the well-known π-calculus [10] with specific operators adapted to correlate with local types. Session types are designed to abstractly capture the structure of (even multi-party) communication protocols [2,3]. The literature on session types provides a rich variety of extensions. *Nested* protocols were introduced by [7] as an extension of multi-party session types as defined e.g. in [2,3]. They offer the possibility to define sub-protocols independently of their parent protocols.

It is essentially the notion of nested protocols that led us to believe that session types could be applied to capture properties of distributed algorithms, especially the so-called round-based distributed algorithms. The latter are typically

structured by a repeated execution of communication patterns by n distributed partners. Often such a pattern involves an exposed coordinator role, whose incarnation may differ from round to round. As such, distributed algorithms very much resemble compositions of nested multi-party communication protocols. Moreover, an essential behavioural property of distributed algorithms is (successful) termination [11,9], despite failures, but it is often elaborate to prove. It turns out that progress (as provided by session types) and termination (as required by distributed algorithms) are closely related. For these reasons, our goal is to apply session type technology to prove behavioural properties of distributed algorithms.

Particularly interesting round-based distributed algorithms were designed in a fault-tolerant way, in order to work in a model where they have to cope with system failures—be it links dropping or manipulating messages, or processes crashing with or without recovery. As the current session type systems are not able to cover fault-tolerance (except for exception handling as in [5,4]), it is necessary to add an appropriate mechanism to cover system failures.

While the detection of conceptual design errors is a standard property of type systems, proving correctness of algorithms despite the occurrence of uncontrollable system failures is not. In the context of distributed algorithms, various kinds of failures have been studied. Often, the correctness of an algorithm does not only depend on the kinds of failures but also of the phase of the algorithm in which they occur, the number of failures, or their likelihood. Here, we only consider a very simple case, namely algorithms that terminate despite arbitrarily many link failures that may occur at any moment in the execution of the algorithm.

Therefore, we extend session types with *optional blocks*, that specify chunks of communication that may at some point fail. This partial communication protocol is protected by the optional block, to ensure that no other process can interfere before the block was resolved and to ensure, that in the case of failure, no parts of the failed communication attempt may influence the further behaviour. In case a link fails, the ambition to guarantee progress requires that the continuation behaviour is not blocked. Therefore, the continuation of an optional block C can be parametrised by a set of values that are either computed by a successful termination of an optional block or are provided beforehand as default values, i.e., we require that for each value that C uses the optional block specifies a default value. An optional block can cover parts of a protocol or even other optional blocks. The type system ensures that communication with optional blocks requires an optional block as communication partner and that only a successful termination of a block releases the protection around its values. The semantics of the calculus then allows us to abort an unguarded optional block at any point. If an optional block models a single communication, its abortion represents a message loss. In summary, optional blocks allow us to automatically derive termination despite arbitrary link failures of distributed algorithms.

Related Work. Type systems are usually designed for scenarios that are free of system failures. An exception is [8] that introduces unreliable broadcast. Within such an unreliable broadcast a transmission can be received by multiple receivers but not necessarily all available receivers. In the latter case, the receiver is

deadlocked. In contrast, we consider failure-tolerant unicast, i.e., communications between a single sender and a single receiver, where in the case of a failure the receiver is not deadlocked but continues using default values.

[5,4] extends session types with exceptions thrown by processes within TRY-and-CATCH-blocks. Both concepts—TRY-and-CATCH-blocks and optional blocks—introduce a way to structurally and semantically encapsulate an unreliable part of a protocol and provide some means to 'detect' a failure and 'react' to it. They are, however, conceptionally and technically different. An obvious difference is the limitation of the inner part of optional blocks towards the computation of values. More fundamentally these approaches differ in the way they allow to 'detect' failures and to 'react' to them.

Optional blocks are designed for the case of system errors that may occur non-deterministically and not necessarily reach the whole system or not even all participants of an optional block, whereas TRY-and-CATCH-blocks model controlled interruption requested by a participant. Hence these approaches differ in the source of an error; raised by the underlying system structure or by a participant. Technically this means that in the presented case failures are introduced by the semantics, whereas in [4] failures are modelled explicitly as THROW-operations. In particular we do not specify, how a participant 'detects' a failure. Different system architectures might provide different mechanisms to do so, e.g. by time-outs. As it is the standard for the analysis of distributed algorithms, our approach allows to port the verified algorithms on different systems architectures, provided that the respective structure and its failure pattern preserves correctness of the considered properties.

The main difference between these two approaches is how they react to failures. In [4] THROW-messages are propagated among nested TRY-and-CATCH-blocks to ensure that all participants are consistently informed about concurrent THROWS of exceptions. In distributed systems such a reaction towards a system error is unrealistic. Distributed processes usually do not have any method to observe an error on another system part and if a participant is crashed or a link fails permanently there is usually no way to inform a waiting communication partner. Instead abstractions (failure detectors) are used to model the detection of failures that can e.g. be implemented by time-outs. Here it is crucial to mention that failure detectors are usually considered to be local and can not ensure global consistency. Distributed algorithms have to deal with the problem that some part of a system may consider a process/link as crashed, while at the same time the same process/link is regarded as correct by another part. This is one of the most challenging problems in the design and verification of distributed algorithms.

In the case of link failures, if a participant is directly influenced by a failure on some other system part (a receiver of a lost message) it will eventually abort the respective communication attempt. If a participant does not depend (the sender in an unreliable link) it may never know about the failure or its nature. Distributed algorithms usually deal with unexpected failures that are hard to detect and often impossible to propagate. Generating correct algorithms for this scenario is difficult and error-prone, thus we need methods to verify them.

Overview. We present global types and restriction in §2, local types and projection in §3, and the session calculus in §4 with a mechanism to check types in §5. In §6 we discuss the properties of the type system. We conclude with §7. The missing proofs and some additional material can be found in [1].

2 Global Types with Optional Blocks

Throughout the paper we use G for global types, T for local types, l for communication labels, s, k for session names, a for shared channels, r for role identifiers, and v for values of a base type (e.g. integer or string). x, y are variables to represent e.g. session names, shared channels, or values. We formally distinguish between roles, labels, process variables, type variables, and names—additionally to identifiers for global/local types, protocols, processes, ... Formally we do however not further distinguish between different kinds of names but use different identifiers $(a, s, v, \ldots)$ to provide hints on the main intended purpose at the respective occurrence. Roles and participants are used as synonyms. To simplify the presentation, we adapt set-like notions for tuples. For example we write $x_i \in \tilde{x}$ if $\tilde{x} = (x_1, \ldots, x_n)$ and $1 \leq i \leq n$. We use $\cdot$ to denote the empty tuple.

Global types describe protocols from a global point of view on systems by interactions between roles. They are used to formalise specifications that describe the desired properties of a system. We extend the basic global types as used e.g. in [2,3] with a global type for optional blocks.

Definition 1 (Global Types). *Global types with optional blocks are given by*

$$G ::= \mathsf{r}_1 \to \mathsf{r}_2 : \sum_{i \in I} \left\{ \, l_i\left(\tilde{x}_i : \tilde{\mathrm{S}}_i\right).G_i \, \right\} \quad | \quad \mathtt{opt}\left\langle \widetilde{\mathsf{r}, \tilde{x} : \tilde{\mathrm{S}}} \mid G \right\rangle.G'$$
$$| \quad G_1 \oplus^{\mathsf{r}} G_2 \quad | \quad G_1 \,||\, G_2 \quad | \quad \mu \mathrm{t}.G \quad | \quad \mathrm{t} \quad | \quad \mathtt{end}$$

$\mathsf{r}_1 \to \mathsf{r}_2 : \sum_{i \in I} \left\{ \, l_i\left(\tilde{x}_i : \tilde{\mathrm{S}}_i\right).G_i \, \right\}$ is the standard way to specify a communication from role r_1 to role r_2, where r_1 has a direct choice between several labels l_i proposed by r_2. Each branch expects values $\tilde{x}_i$ of sorts $\tilde{\mathrm{S}}_i$ and executes the continuation G_i. When I is a singleton, we write $\mathsf{r}_1 \to \mathsf{r}_2 : l\left(\tilde{x} : \tilde{\mathrm{S}}\right)$. $G_1 \oplus^{\mathsf{r}} G_2$ introduces so-called located (or internal) choice: the choice for one role r between two distinct protocol branches. The parallel composition $G_1 \,||\, G_2$ allows to specify independent parts of a protocol. $\mu \mathrm{t}.G$ and t are used to allow for recursion. `end` denotes the successful completion. We often omit trailing `end` clauses.

We add $\mathtt{opt}\left\langle \widetilde{\mathsf{r}, \tilde{x} : \tilde{\mathrm{S}}} \mid G \right\rangle.G'$ to describe an optional block between the roles $\mathsf{r}_1, \ldots, \mathsf{r}_n$, where $\widetilde{\mathsf{r}, \tilde{x} : \tilde{\mathrm{S}}}$ abbreviates the sequence $\mathsf{r}_1, \tilde{x}_1 : \tilde{\mathrm{S}}_1, \ldots, \mathsf{r}_n, \tilde{x}_n : \tilde{\mathrm{S}}_n$. Here G is the protocol that is encapsulated by the optional block and the $\tilde{x}_i$ are so-called default values that are used within the continuation G' of the surrounding parent session if the optional block fails. There is one (possibly empty) vector of default values $\tilde{x}_i$ for each role r_i. The inner part G of an optional block is a protocol that (in the case of success) is used to compute the vectors of return values. The

typing rules ensure that for each role r_i the type of the computed vector coincides with the type $\tilde{\mathrm{S}}_i$ of the specified vector of default values $\tilde{x}_i$. Intuitively, if the block does not fail, each participant can use its respective vector of computed values in the continuation G'. Otherwise, the default values are used.

Optional blocks capture the main features of a failure very naturally: a part of a protocol either succeeds or fails. They also encapsulate the source and direct impact of the failure, which allows us to study their implicit effect—as e.g. missing communication partners—on the overall behaviour of protocols. With that they help us to specify, implement, and verify failure-tolerant algorithms.

Using optional blocks we provide a natural and simple specification of an unreliable link c between the two roles src and trg, where in the case of success the value v_{src} is transmitted and in the case of failure a default value v_{trg} is used by the receiver.

Example 1 (Global Type of an Unreliable Link).

$$G_{\mathsf{UL}}(\mathsf{src}, v_{\mathsf{src}}; \mathsf{trg}, v_{\mathsf{trg}}) = \mathtt{opt}\langle \mathsf{src}, \cdot, \mathsf{trg}, v_{\mathsf{trg}} : \mathrm{V} \mid (\mathsf{src} \to \mathsf{trg} : c(v_{\mathsf{src}} : \mathrm{V}).\mathtt{end})\rangle$$

Here we have a single communication step—to model the potential loss of a single message—that is covered within an optional block. In the term $G_{\mathsf{UL}}(\mathsf{src}, v_{\mathsf{src}}; \mathsf{trg}, v_{\mathsf{trg}}).G'$ the receiver trg may use the transmitted value v_{src} in the continuation G' if the communication succeeds or else uses its default value v_{trg}. Note that the optional block above specifies the empty sequence of values as default values for the sending process src, i.e., the sender needs no default values.

Well-Formed. Following [7] we type all objects appearing in global types with *kinds* (types for types) $\mathrm{K} ::= \mathrm{Role} \mid \mathrm{Val}$. Val are value-kinds, which are first-order types for values (like $\mathbb{B}$ for boolean) or data types. Role is used for identifiers of roles. We adopt the definition of *well-kinded* global types from [7] that basically ensures that all positions $\mathsf{r}, \mathsf{r}_1, \mathsf{r}_2, \tilde{\mathsf{r}}_1, \tilde{\mathsf{r}}_2$ in global types can be instantiated only by objects of type Role. According to [7] a global type G is *projectable* if for each occurrence of $G_1 \oplus^{\mathsf{r}} G_2$ in the type and for any free role $\mathsf{r}' \neq \mathsf{r}$ we have $G_1|_{\mathsf{r}'} = G_2|_{\mathsf{r}'}$. Additionally we require (similar to sub-sessions in [7]) for a global type G to be projectable that, for each optional block $\mathtt{opt}\left\langle \widetilde{\mathsf{r}, \tilde{x} : \tilde{\mathrm{S}} \mid G_1} \right\rangle.G_2$ in G, all roles in G_1 are contained in $\tilde{\mathsf{r}}$. A global type is *well-formed* when it is well-kinded and projectable, and satisfies the standard linearity condition [2]. For more intuition on the notion of well-formedness and examples for non-well-formed protocols we refer to [7]. In the examples, we use V as the type of values.

3 Local Types with Optional Blocks

Local types describe a local and partial point of view on a global communication protocol w.r.t. a single participant. They are used to validate and monitor distributed programs. We extend the basic local types as used e.g. in [2,3] with a local type for optional blocks.

Definition 2 (Local Types). *Local types with optional blocks are given by*

$$T ::= \texttt{get}[\mathsf{r}]?_{i\in I}\left\{\, l_i\left(\tilde{x}_i{:}\tilde{\mathrm{S}}_i\right).T_i \,\right\} \quad | \quad \texttt{send}[\mathsf{r}]!_{i\in I}\left\{\, l_i\left(\tilde{x}_i{:}\tilde{\mathrm{S}}_i\right).T_i \,\right\}$$
$$| \quad \texttt{opt}[\tilde{\mathsf{r}}]\langle T\rangle\left(\tilde{x}{:}\tilde{\mathrm{S}}\right).T' \quad | \quad T_1 \oplus T_2 \quad | \quad T_1 \,||\, T_2 \quad | \quad \mu \mathrm{t}.T \quad | \quad \mathrm{t} \quad | \quad \texttt{end}$$

The first two operators specify endpoint primitives for communications with `get` for the receiver side—where r is the sender—and `send` for the sender side—where r denotes the receiver. Accordingly, they introduce the two possible local views of a global type for communication. $T_1 \oplus T_2$ is the local view of the global type $G_1 \oplus^{\mathsf{r}} G_2$ for a choice determined by the role r for which this local type is created. $T_1 \,||\, T_2$ represents the local view of the global type for parallel composition, i.e., describes independent parts of the protocol for the considered role. Again $\mu \mathrm{t}.T$ and t are used to introduce recursion and `end` denotes the successful completion of a protocol. Again we usually omit trailing `end` clauses.

We add the local type $\texttt{opt}[\tilde{\mathsf{r}}]\langle T\rangle\left(\tilde{x}{:}\tilde{\mathrm{S}}\right).T'$. It initialises an optional block between the roles $\tilde{\mathsf{r}}$ around the local type T, where the currently considered participant r (called *owner*) is a participant of this block, i.e., $\mathsf{r} \in \tilde{\mathsf{r}}$. After the optional block the local type continues with T'.

Projection. To ensure that a global type and its local types coincide, global types are projected to their local types. Accordingly we define the projection $(G)\Downarrow_{\mathsf{r}_p}$ of a global type G on a role r_p for the case that G describes an optional block.

$$\left(\texttt{opt}\left\langle \widetilde{\mathsf{r}, \tilde{x}{:}\tilde{\mathrm{S}} \mid G}\right\rangle.G'\right)\Downarrow_{\mathsf{r}_p} = \begin{cases} \texttt{opt}[\tilde{\mathsf{r}}]\left\langle G\Downarrow_{\mathsf{r}_p}\right\rangle\left(\tilde{x}_i{:}\tilde{\mathrm{S}}_i\right).(G')\Downarrow_{\mathsf{r}_p} & \text{if } \mathsf{r}_p = \mathsf{r}_i \in \tilde{\mathsf{r}} \\ & \quad \text{and } \tilde{x}_i \neq \cdot \\ \texttt{opt}[\tilde{\mathsf{r}}]\left\langle G\Downarrow_{\mathsf{r}_p}\right\rangle(\cdot) \;||\; (G')\Downarrow_{\mathsf{r}_p} & \text{if } \mathsf{r}_p = \mathsf{r}_i \in \tilde{\mathsf{r}} \\ & \quad \text{and } \tilde{x}_i = \cdot \\ G'\Downarrow_{\mathsf{r}_p} & \text{else} \end{cases}$$

The projection rule for optional blocks has three cases. The last case is used to skip optional blocks when they are projected to roles that do not participate. The first two cases handle projection of optional blocks to one of its participants. A local optional block is generated with the projection of G as content.

The first two cases check whether the optional block indeed computes any values for the role we project onto. They differ only in the way that the continuation of the optional block and its inner part are connected. If the projected role does not specify default values—because no such values are required—the projected continuation $(G')\Downarrow_{\mathsf{r}_p}$ can be placed in parallel to the optional block (second case). Otherwise, the continuation has to be guarded by the optional block and, thus, by the computation of the computed values (first case).

By distinguishing between these two first cases, we follow the same line of argument as used for sub-sessions in [7], where the projected continuation of a sub-session `call` is either in parallel to the projection of the `call` itself or connected sequentially. Intuitively, whenever the continuation depends on the outcome of the optional block it has to be connected sequentially. A complete list of all projection rules can be found in [1].

Example 2 (Projection of Unreliable Links).

$$\begin{aligned} G_{\mathsf{UL}}(\mathsf{src}, v_{\mathsf{src}}; \mathsf{trg}, v_{\mathsf{trg}}) \Downarrow_{\mathsf{src}} &= T_{\mathsf{UL}\uparrow}(\mathsf{src}, v_{\mathsf{src}}, \mathsf{trg}) \\ &= \mathtt{opt}[\mathsf{scr}, \mathsf{trg}]\langle \mathtt{send}[\mathsf{trg}]!c(v_{\mathsf{src}} : \mathrm{V})\rangle(\cdot) \\ G_{\mathsf{UL}}(\mathsf{src}, v_{\mathsf{src}}; \mathsf{trg}, v_{\mathsf{trg}}) \Downarrow_{\mathsf{trg}} &= T_{\mathsf{UL}\downarrow}(\mathsf{src}, v_{\mathsf{src}}; \mathsf{trg}, v_{\mathsf{trg}}) \\ &= \mathtt{opt}[\mathsf{src}, \mathsf{trg}]\langle \mathtt{get}[\mathsf{src}]?c(v_{\mathsf{src}} : \mathrm{V})\rangle(v_{\mathsf{trg}} : \mathrm{V}) \end{aligned}$$

When projected onto its sender, the global type for a communication over an unreliable link results in the local type $T_{\mathsf{UL}\uparrow}(\mathsf{src}, v_{\mathsf{src}}, \mathsf{trg})$ that consists of an optional block containing a send operation towards trg. Since the optional block for the sender does not specify any default values, the local type $T_{\mathsf{UL}\uparrow}(\mathsf{src}, v_{\mathsf{src}}, \mathsf{trg})$ will be placed in parallel to the projection of the continuation. The projection onto the receiver results in the local type $T_{\mathsf{UL}\downarrow}(\mathsf{src}, v_{\mathsf{src}}; \mathsf{trg}, v_{\mathsf{trg}})$ that consists of an optional block containing a receive operation from src. Here a default value is necessary for the case that the message is lost. So the type $T_{\mathsf{UL}\downarrow}(\mathsf{src}, v_{\mathsf{src}}; \mathsf{trg}, v_{\mathsf{trg}})$ has to be composed sequentially with the projection of the continuation.

4 A Session Calculus with Optional Blocks

Global types (and the local types that are derived from them) can be considered as specifications that describe the desired properties of the system we want to analyse. The process calculus, that we use to model/implement the system, is in the case of session types usually a variant of the π-calculus [10]. We extend a basic session-calculus as used e.g. in [2,3] with two operators.

Definition 3 (Processes). *Processes are given by*

$$\begin{aligned} P ::= {} & a(\tilde{x}).P \mid \overline{a}\langle \tilde{s} \rangle.P \mid k?[\mathsf{r}_1, \mathsf{r}_2]_{i \in I} \{\, l_i(\tilde{x}_i).P_i \,\} \mid k![\mathsf{r}_1, \mathsf{r}_2] l\langle \tilde{v} \rangle.P \\ \mid {} & \mathtt{opt}[\mathsf{r}; \tilde{v}; \tilde{\mathsf{r}}]\langle P \rangle(\tilde{x}).P' \mid [\mathsf{r}]\langle \tilde{v} \rangle \\ \mid {} & (\nu x)\, P \mid P_1 + P_2 \mid P_1 \mid P_2 \mid \mu \mathrm{X} : P \mid \mathrm{X} \mid \mathbf{0} \end{aligned}$$

The prefixes $a(\tilde{x}).P$ and $\overline{a}\langle \tilde{s} \rangle.P$ are inherited from the π-calculus and are used for external invitations. Using the shared channel a, an external participant can be invited with the output $\overline{a}\langle \tilde{s} \rangle.P$ transmitting the session channels $\tilde{s}$ that are necessary to participate and the external participant can accept the invitation using the input $a(\tilde{x}).P$. The following two operators introduce a branching input and the corresponding transmission on the session channel k from r_1 to r_2. These two operators correspond to the local types for `get` and `send`. Restriction $(\nu x)\, P$ allows to generate a fresh name that is not known outside of the scope of this operator unless it was explicitly communicated. For simplicity and following [6] we assume that only shared channels a for external invitations and session channels s, k for not yet initialised sub-sessions are restricted, because this covers the interesting cases[1] and simplifies the typing rules in Figure 2. The term $P_1 + P_2$ either behaves as P_1 or P_2. $P_1 \mid P_2$ defines the parallel composition of

[1] Sometimes it might be useful to allow the restriction of values, e.g. for security. For this case an additional restriction operator can be introduced.

the processes P_1 and P_2. $\mu \mathrm{X}\colon P$ and X are used to introduce recursion. $\mathbf{0}$ denotes the successful completion.

To implement optional blocks, we add $\mathtt{opt}[\mathsf{r}; \tilde{v}_d; \tilde{\mathsf{r}}]\langle P \rangle(\tilde{x}).P'$ and $[\mathsf{r}]\langle \tilde{v} \rangle$. The former defines an optional block between the roles $\tilde{\mathsf{r}}$ around the process P with the default values $\tilde{v}_d$. We require that the owner r of this block is one of its participants $\tilde{\mathsf{r}}$, i.e., $\mathsf{r} \in \tilde{\mathsf{r}}$. In the case of success, $[\mathsf{r}]\langle \tilde{v} \rangle$ transmits the computed values $\tilde{v}$ from within the optional block to the continuation P' to be substituted for the variables $\tilde{x}$ within P'. If the optional block fails the variables $\tilde{x}$ of P' are replaced by the default values $\tilde{v}_d$ instead. Without loss of generality we assume that the roles $\tilde{\mathsf{r}}$ of optional blocks are distinct. Since optional blocks can compute only values and their defaults need to be of the same kind, $[\mathsf{r}]\langle \tilde{v} \rangle$ and the defaults cannot carry session names, i.e., names used as session channels. The type system ensures that the inner part P of a successful optional block reaches some $[\mathsf{r}]\langle \tilde{v} \rangle$ and thus transmits computed values of the expected kinds in exactly one of its parallel branches. The semantics presented below ensures that every optional block can transmit at most one vector of computed values and has to fail otherwise. Similarly optional blocks, that use roles in their inner part P that are different from $\tilde{r}$ and are not newly introduced as part of a sub-session within P, cannot be well-typed. Since optional blocks open a context block around their inner part that separates P from the continuation P', scopes as introduced by input prefixes and restriction that are opened within P cannot cover parts of P'.

Example 3 (Implementation of Unreliable Links).

$$P_{\mathsf{UL}\uparrow}(\mathsf{p}_1, v_1, \mathsf{p}_2) = \mathtt{opt}[\mathsf{p}_1; \cdot; \mathsf{p}_1, \mathsf{p}_2]\langle s![\mathsf{p}_1, \mathsf{p}_2]c\langle v_1 \rangle.\, [\mathsf{p}_1]\langle \cdot \rangle \rangle(\cdot)$$
$$P_{\mathsf{UL}\downarrow}(\mathsf{p}_1, \mathsf{p}_2, v_2) = \mathtt{opt}[\mathsf{p}_2; v_2; \mathsf{p}_1, \mathsf{p}_2]\langle s?[\mathsf{p}_1, \mathsf{p}_2]c(x).\, [\mathsf{p}_2]\langle x \rangle \rangle(y)$$

$P_{\mathsf{UL}\uparrow}(\mathsf{p}_1, v_1, \mathsf{p}_2)$ is the implementation of a single send action on an unreliable link and $P_{\mathsf{UL}\downarrow}(\mathsf{p}_1, \mathsf{p}_2, v_2)$ the corresponding receive action. Here a continuation of the sender cannot gain any information from the modelled communication; not even whether it succeeded, whereas a continuation of the receiver in the case of success obtains the transmitted value v_1 and else its own default value v_2.

Again we usually omit trailing $\mathbf{0}$. In Definition 3 all occurrences of x, $\tilde{x}$, and $\tilde{x}_i$ refer to bound names of the respective operators. The set $\mathrm{FN}(P)$ of free names of P is the set of names of P that are not bound. A substitution $\{{}^{y_1}/_{x_1}, \ldots, {}^{y_n}/_{x_n}\} = \{{}^{\tilde{y}}/_{\tilde{x}}\}$ is a finite mapping from names to names, where the $\tilde{x}$ are pairwise distinct. The application of a substitution on a term $P\{{}^{\tilde{y}}/_{\tilde{x}}\}$ is defined as the result of simultaneously replacing all free occurrences of x_i by y_i, possibly applying alpha-conversion to avoid capture or name clashes. For all names $n \notin \tilde{x}$ the substitution behaves as the identity mapping. We use '.' (as e.g. in $a(\tilde{x}).P$) to denote sequential composition. In all operators the part before '.' guards the continuation after the '.', i.e., the continuation cannot reduce before the guard was reduced. A subprocess of a process is *guarded* if it occurs after such a guard, i.e., is the continuation (or part of the continuation) of a guard. Guarded subprocesses can be *unguarded* by steps that remove the guard. We identify processes up to a standard variant of structural congruence defined in [1].

$$(\mathsf{comS})\ \frac{j \in I}{E[k![\mathsf{r}_1,\mathsf{r}_2]l_j\langle\tilde{v}\rangle.P \mid k?[\mathsf{r}_1,\mathsf{r}_2]_{i\in I}\{\ l_i(\tilde{x}_i).P_i\ \}] \longmapsto E[P \mid P_j\{\tilde{v}/\tilde{x}_j\}]}$$

$$(\mathsf{choice})\ \frac{P_i \longmapsto P_i'}{E[P_1 + P_2] \longmapsto E[P_i']} \qquad (\mathsf{comC})\ \frac{}{E[\overline{a}\langle\tilde{s}\rangle.P_1 \mid a(\tilde{x}).P_2] \longmapsto E[P_1 \mid P_2\{\tilde{s}/\tilde{x}\}]}$$

$$(\mathsf{fail})\ \frac{}{E[\mathtt{opt}[\mathsf{r};\tilde{v}_d;\tilde{\mathsf{r}}]\langle P\rangle(\tilde{x}).P'] \longmapsto E[P'\{\tilde{v}_d/\tilde{x}\}]}$$

$$(\mathsf{succ})\ \frac{}{E[\mathtt{opt}[\mathsf{r};\tilde{v}_d;\tilde{\mathsf{r}}]\langle[\mathsf{r}]\langle\tilde{v}\rangle\rangle(\tilde{x}).P] \longmapsto E[P\{\tilde{v}/\tilde{x}\}]}$$

$$(\mathsf{cSO})\ \frac{j \in I \quad \mathrm{roles}(C_{\mathsf{opt}}) \doteq \mathrm{roles}(C'_{\mathsf{opt}}) \quad \mathrm{owner}(C_{\mathsf{opt}}) = \mathsf{r}_1 \quad \mathrm{owner}(C'_{\mathsf{opt}}) = \mathsf{r}_2}{\begin{array}{l} E\big[E_{\mathsf{R}}[C_{\mathsf{opt}}[k![\mathsf{r}_1,\mathsf{r}_2]l_j\langle\tilde{v}\rangle.P]] \mid E'_{\mathsf{R}}\big[C'_{\mathsf{opt}}[k?[\mathsf{r}_1,\mathsf{r}_2]_{i\in I}\{\ l_i(\tilde{x}_i).P_i\ \}]\big]\big] \\ \quad \longmapsto E\big[E_{\mathsf{R}}[C_{\mathsf{opt}}[P]] \mid E'_{\mathsf{R}}\big[C'_{\mathsf{opt}}[P_j\{\tilde{v}/\tilde{x}_j\}]\big]\big] \end{array}}$$

$$(\mathsf{cCO})\ \frac{\mathrm{roles}(C_{\mathsf{opt}}) \doteq \mathrm{roles}(C'_{\mathsf{opt}})}{E\big[E_{\mathsf{R}}[C_{\mathsf{opt}}[\overline{a}\langle\tilde{s}\rangle.P_1]] \mid E'_{\mathsf{R}}\big[C'_{\mathsf{opt}}[a(\tilde{x}).P_2]\big]\big] \longmapsto E\big[E_{\mathsf{R}}[C_{\mathsf{opt}}[P_1]] \mid E'_{\mathsf{R}}\big[C'_{\mathsf{opt}}[P_2\{\tilde{s}/\tilde{x}\}]\big]\big]}$$

Fig. 1. Reduction Rules

Reduction Semantics. In [7] the semantics is given by a set of reduction rules that are defined w.r.t. evaluation contexts. We extend them with optional blocks.

Definition 4. $E ::= [\,] \quad | \quad P \mid E \quad | \quad (\nu x)\,E \quad | \quad \mathtt{opt}[\mathsf{r};\tilde{v};\tilde{\mathsf{r}}]\langle E\rangle(\tilde{x}).P'$

Intuitively an evaluation context is a term with a single hole that is not guarded. Additionally, we introduce two variants of evaluation contexts and a context for blocks that are used to simplify the presentation of our new rules.

Definition 5. $E_{\mathsf{R}} ::= [\,] \quad | \quad P \mid E_{\mathsf{R}} \quad | \quad \mathtt{opt}[\mathsf{r};\tilde{v};\tilde{\mathsf{r}}]\langle E_{\mathsf{R}}\rangle(\tilde{x}).P'$
$C_{\mathsf{opt}} ::= \mathtt{opt}[\mathsf{r};\tilde{v};\tilde{\mathsf{r}}]\langle E_{\mathsf{P}}\rangle(\tilde{x}).P'$, *where* $E_{\mathsf{P}} ::= [\,] \quad | \quad P \mid E_{\mathsf{P}}$

Accordingly, a C_{opt}-context consists of exactly one optional block that contains an E_{P}-context, i.e., a single hole that can occur within the parallel composition of arbitrary processes. We define the function $\mathrm{roles}(\mathtt{opt}[\mathsf{r};\tilde{v};\tilde{\mathsf{r}}]\langle E_{\mathsf{P}}\rangle(\tilde{x}).P') \triangleq \tilde{\mathsf{r}}$, to return the roles of the optional block of a C_{opt}-context, and the function $\mathrm{owner}(\mathtt{opt}[\mathsf{r};\tilde{v};\tilde{\mathsf{r}}]\langle E_{\mathsf{P}}\rangle(\tilde{x}).P') \triangleq \mathsf{r}$, to return its owner.

Figure 1 presents the reduction rules. The Rules (comS), (choice), and (comC) deal with the standard operators for communication, choice, and external invitations to sessions, respectively. Since evaluation contexts E contain optional blocks, these rules allow for steps within a single optional block. To capture optional blocks, we introduce the new Rules (fail), (succ), (cSO), and (cCO). Here $\doteq$ means that the two compared vectors contain the same roles but not necessarily in the same order, i.e., $\doteq$ checks whether the set of participants of two optional blocks are the same. The Rules (comS) and (comC) represent two different kinds of communication. They define communications within a session and external session invitations, respectively. In both cases communication is an axiom that requires the occurrence of two matching counterparts of communication primitives (of the respective kind) to be placed in parallel within an

evaluation context. As a consequence of the respective communication step the continuations of both communication primitives are unguarded and the values transmitted in the communication step are instantiated (substituted) in the receiver continuation. (choice) allows the reduction of either side of a choice, if the respective side can perform a step.

The two rules (succ) and (fail) describe the main features of optional blocks, namely how they succeed (succ) and what happens if they fail (fail). (fail) aborts an optional block, i.e., removes it and unguards its continuation instantiated with the default values. This rule can be applied whenever an optional block is unguarded, i.e., there is no way to ensure that an optional block does indeed perform any step. In combination with (succ), it introduces the non-determinism that is used to express the random nature in that system errors may occur.

(succ) is the counterpart of (fail); it removes a successfully completed optional block and unguards its continuation instantiated with the computed results. To successfully complete an optional block, we require that its content has to reduce to a single occurrence of $[\mathsf{r}]\langle\tilde{v}\rangle$, where r is the owner of the block and accordingly one of the participating roles. Since (succ) and (fail) are the only ways to reduce $[\mathsf{r}]\langle\tilde{v}\rangle$, this ensures that a successful optional block can compute only a single vector of return values. Other parallel branches in the inner part of an optional block have to terminate with $\mathbf{0}$. This ensures that no confusion can arise from the computation of different values in different parallel branches. Since at the process-level an optional block covers only a single participant, this limitation does not restrict the expressive power of the considered processes. If the content of an optional block cannot reduce to $[\mathsf{r}]\langle\tilde{v}\rangle$ the optional block is doomed to fail.

The remaining rules describe how different optional blocks can interact. Here, we need to ensure that communication from within an optional block ensures isolation, i.e., that such communications are restricted to the encapsulated parts of other optional blocks. The E_{R}-contexts allow for two such blocks to be nested within different optional blocks. The exact definition of such a communication rule depends on the semantics of the considered calculi and their communication rules. Here there are the Rules (cSO) and (cCO). They are the counterparts of (comS) and (comC) and accordingly allow for the respective kind of communication step. As an example consider Rule (cSO). In comparison to (comS), Rule (cSO) ensures that communications involving the content of an optional block are limited to two such contents of optional blocks with the same participants. This ensures that optional blocks describe the local view-points of the encapsulated protocol.

Optional blocks do not allow for scope extrusion of restricted names, i.e., a name restricted within an optional block cannot be transmitted nor can an optional block successfully be terminated if the computed result values are subject to a restriction from the content of the optional block. Also values that are communicated between optional blocks can be used only by the continuation of the optional block and only if the optional block was completed successfully. If an optional block fails while another process is still waiting for a communication within its optional block, the latter optional block is doomed to fail. Note that the semantics of optional blocks is inherently synchronous, since an

optional sending operation can realise the failing of its matching receiver (e.g. by $\mathtt{opt}[\mathsf{r}_1; \mathit{fail}; \mathsf{r}_2]\langle \ldots [\mathsf{r}_1]\langle ok \rangle\rangle(x).P$). Let $\longmapsto^+$ denote the transitive closure of $\longmapsto$ and let $\longmapsto^*$ denote the reflexive and transitive closure of $\longmapsto$, respectively.

5 Well-Typed Processes

Now we connect types with processes by the notion of well-typedness. A process P is *well-typed* if it satisfies a typing judgement of the form $\Gamma \vdash P \triangleright \Delta$, i.e., under the *global environment* Γ, P is validated by the *session environment* Δ. We extend environments defined in [7] with a primitive for session environments.

Definition 6 (Environments).

$$\Gamma ::= \emptyset \quad | \quad \Gamma, a : T[\mathsf{r}] \quad | \quad \Gamma, s : G$$
$$\Delta ::= \emptyset \quad | \quad \Delta, s[\mathsf{r}] : T \quad | \quad \Delta, s[\mathsf{r}]^\bullet : T \quad | \quad \Delta, \mathsf{r} : \tilde{\mathrm{S}}^\uparrow$$

The global environment Γ relates shared channels to the type of the invitation they carry and session channels s to the global type G they implement. $a : T[\mathsf{r}]$ means that a is used to send and receive invitations to play role r with local type T. The session environment Δ relates pairs of session channels s and roles r to local types T, where $s[\mathsf{r}]^\bullet : T$ denotes the permission to transmit the corresponding s. We add the declaration $\mathsf{r} : \tilde{\mathrm{S}}^\uparrow$, to cover the kinds of the return values of an optional block of the owner r. A session environment is *closed* if it does not contain declarations $\mathsf{r} : \tilde{\mathrm{S}}^\uparrow$. We assume that initially session environments do not contain declarations $\mathsf{r} : \tilde{\mathrm{S}}^\uparrow$, i.e., are closed. Such declarations are introduced while typing the content of an optional block. Whereby the typing rules ensure that environments can never contain more than one declaration $\mathsf{r} : \tilde{\mathrm{S}}^\uparrow$.

Let $(\Delta, s[\mathsf{r}] : \mathtt{end}) = \Delta$. If $s[\mathsf{r}]$ does not appear in Δ, we write $\Delta(s[\mathsf{r}]) = 0$. Following [7] we assume an operator $\otimes$ such that (1) $\Delta \otimes \emptyset = \Delta$, (2) $\Delta_1 \otimes \Delta_2 = \Delta_2 \otimes \Delta_1$, (3) $\Delta_1 \otimes \left(\Delta_2, \mathsf{r} : \tilde{\mathrm{S}}^\uparrow\right) = \left(\Delta_1, \mathsf{r} : \tilde{\mathrm{S}}^\uparrow\right) \otimes \Delta_2$, (4) $\Delta_1 \otimes (\Delta_2, s[\mathsf{r}] : T) = (\Delta_1, s[\mathsf{r}] : T) \otimes \Delta_2$ if $\Delta_1(s[\mathsf{r}]) = 0 = \Delta_2(s[\mathsf{r}])$, and (5) $(\Delta_1, s[\mathsf{r}] : T_1) \otimes (\Delta_2, s[\mathsf{r}] : T_2) = (\Delta_1, s[\mathsf{r}] : T_1 \mid\mid T_2) \otimes \Delta_2$. Thus $\otimes$ allows to split parallel parts of local types. We write $\vdash v : \mathrm{S}$ if value v is of kind S.

In Figure 2 we extend the typing rules of [7] with the Rules (Opt) and (OptE) for optional blocks. (Opt) ensures that (1) the process and the local type specify the same set of roles $\tilde{\mathsf{r}} \doteq \tilde{\mathsf{r}}'$ as participants of the optional block, (2) the kinds of the default values $\tilde{v}$, the arguments $\tilde{x}$ of the continuation P', and the respective variables $\tilde{y}$ in the local type coincide, (3) the continuation P' is well-typed w.r.t. the part Δ' of the current session environment and the remainder T' of the local type of $s[\mathsf{r}_1]$, (4) the content P of the block is well-typed w.r.t. the session environment $\Delta, s[\mathsf{r}_1] : T, \mathsf{r}_1 : \tilde{\mathrm{S}}^\uparrow$, where $\mathsf{r}_1 : \tilde{\mathrm{S}}^\uparrow$ ensures that P computes return values of the kinds $\tilde{\mathrm{S}}$ if no failure occurs, and (5) the return values of a surrounding optional block cannot be returned in a nested block, because of the condition $\nexists \mathsf{r}'', \tilde{\mathrm{K}}.\ \mathsf{r}'' : \tilde{\mathrm{K}}^\uparrow \in \Delta$. (OptE) ensures that the kinds of the values computed by a successful completion of an optional block match the kinds of the respective default values. Apart from that this rule is similar

$$(\mathsf{I})\,\frac{\Gamma \vdash P \triangleright \Delta, x[\mathsf{r}]:T \quad \Gamma(a) = T[\mathsf{r}]}{\Gamma \vdash a(x).P \triangleright \Delta} \qquad (\mathsf{O})\,\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma(a) = T[\mathsf{r}]}{\Gamma \vdash \overline{a}\langle s\rangle.P \triangleright \Delta, s[\mathsf{r}]^{\bullet}:T} \qquad (\mathsf{N})\,\frac{}{\Gamma \vdash \mathbf{0} \triangleright \emptyset}$$

$$(\mathsf{C})\,\frac{\left(\Gamma \vdash P_i \triangleright \Delta, k[\mathsf{r}_2]:T_i \quad \vdash \tilde{y}_i:\tilde{\mathrm{S}}_i\right)_{i\in I}}{\Gamma \vdash k?[\mathsf{r}_1,\mathsf{r}_2]_{i\in I}\,\{\, l_i(\tilde{y}_i).P_i \,\} \triangleright \Delta, k[\mathsf{r}_2]:\mathtt{get}[\mathsf{r}_1]?_{i\in I}\left\{\, l_i\left(\tilde{x}_i:\tilde{\mathrm{S}}_i\right).T_i \,\right\}}$$

$$(\mathsf{S1})\,\frac{\Gamma \vdash P_1 \triangleright \Delta, s[\mathsf{r}]:T_1 \quad \Gamma \vdash P_2 \triangleright \Delta, s[\mathsf{r}]:T_2}{\Gamma \vdash P_1 + P_2 \triangleright \Delta, s[\mathsf{r}]:T_1 \oplus T_2} \qquad (\mathsf{S2})\,\frac{\Gamma \vdash P \triangleright \Delta, s[\mathsf{r}]:T_i \quad i \in \{1,2\}}{\Gamma \vdash P \triangleright \Delta, s[\mathsf{r}]:T_1 \oplus T_2}$$

$$(\mathsf{S})\,\frac{\Gamma \vdash P \triangleright \Delta, k[\mathsf{r}_1]:T_j \quad \vdash \tilde{v}:\tilde{\mathrm{S}}_j}{\Gamma \vdash k![\mathsf{r}_1,\mathsf{r}_2]l_j\langle\tilde{v}\rangle.P \triangleright \Delta, k[\mathsf{r}_1]:\mathtt{send}[\mathsf{r}_2]!_{i\in I}\left\{\, l_i\left(\tilde{x}_i:\tilde{\mathrm{S}}_i\right).T_i \,\right\}}$$

$$(\mathsf{Pa})\,\frac{\Gamma \vdash P_1 \triangleright \Delta_1 \quad \Gamma \vdash P_2 \triangleright \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \otimes \Delta_2} \qquad (\mathsf{R})\,\frac{\Gamma, x:T[\mathsf{r}] \vdash P \triangleright \Delta}{\Gamma \vdash (\nu x)\,P \triangleright \Delta} \qquad (\mathsf{OptE})\,\frac{\vdash \tilde{v}:\tilde{\mathrm{S}}}{\Gamma \vdash [\mathsf{r}]\langle\tilde{v}\rangle \triangleright \mathsf{r}:\tilde{\mathrm{S}}^{\uparrow}}$$

$$(\mathsf{Opt})\,\frac{\begin{array}{c}\tilde{\mathsf{r}} \doteq \tilde{\mathsf{r}}' \quad \Gamma \vdash P \triangleright \Delta, s[\mathsf{r}_1]:T, \mathsf{r}_1:\tilde{\mathrm{S}}^{\uparrow} \quad \nexists \mathsf{r}'', \tilde{\mathrm{K}}.\ \mathsf{r}'':\tilde{\mathrm{K}}^{\uparrow} \in \Delta \\ \Gamma \vdash P' \triangleright \Delta', s[\mathsf{r}_1]:T' \quad \vdash \tilde{x}:\tilde{\mathrm{S}} \quad \vdash \tilde{v}:\tilde{\mathrm{S}}\end{array}}{\Gamma \vdash \mathtt{opt}[\mathsf{r}_1;\tilde{v};\tilde{\mathsf{r}}]\langle P\rangle(\tilde{x}).P' \triangleright \Delta \otimes \Delta', s[\mathsf{r}_1]:\mathtt{opt}[\tilde{\mathsf{r}}']\langle T\rangle\left(\tilde{y}:\tilde{\mathrm{S}}\right).T'}$$

Fig. 2. Typing Rules

to (N) in Figure 2. Since (OptE) is the only way to consume an instance of $\mathsf{r}:\tilde{\mathrm{S}}^{\uparrow}$, this rule checks that—ignoring the possibility to fail—the content of an optional block reduces to $[\mathsf{r}]\langle\tilde{v}\rangle$, if the corresponding local type requires it to do so. Combining these rules, (Opt) introduces exactly one occurrence of $\mathsf{r}:\tilde{\mathrm{S}}^{\uparrow}$ in the session environment, the function $\otimes$ in (Pa) for parallel processes in Figure 2 ensures that this occurrence reaches exactly one of the parallel branches of the content of the optional block, and finally only (OptE) allows to terminate a branch with this occurrence. This ensures that—ignoring the possibility to fail—each block computes exactly one vector of return values $[\mathsf{r}]\langle\tilde{v}\rangle$ (or, more precisely, one such vector for each choice-branch). For an explanation of the remaining rules we refer to [2,3] and [7]. Applying these typing rules is elaborate but straightforward and can be automated easily, since for all processes except choice exactly one rule applies and all parameters except for restriction are determined by the respective process. Thus, the number of different proof-trees is determined by the number of choices and the type of restricted channels can be derived using back-tracking.

6 Properties of the Type Systems

Subject reduction is a basic property of each type system. It is this property that allows us to reason statically about terms, by ensuring that whenever a process its well-typed then all its derivatives are well-typed as well. Hence, for all properties the type system ensures for well-typed terms, it is not necessary to compute executions but only to test for well-typedness of the original term. We use a strong variant of subject reduction that additionally involves the condition

$\Delta \mapsto \Delta'$, in order to capture how the local types evolve alongside the reduction of processes. Therefore the effect of reductions on processes on the corresponding local types is captured within $\mapsto$ that can be found in [1].

Following [7] we use coherence to prove progress and completion. A session environment is *coherent* if it is composed of the projections of well-formed global types. Most of the reduction rules preserve coherence. The failing of optional blocks can however temporary invalidate this property. A failing optional block is not a problem for the process itself, because the continuation of the process is instantiated with the default value and this process with a corresponding session environment corresponds to the projection of the global type of the continuation. But a failing optional block may cause another part of the network, i.e., a parallel process, to lose coherence. If another, parallel optional block is waiting for a communication with the former, it is doomed to fail. This situation of a single optional block without its dual communication partner cannot result from the projection of a global type. Due to the interleaving of steps, an execution starting in a process with a coherent session environment may lead to a state in which there are several single optional blocks at the same time. However, coherence ensures that for all such reachable processes there is a finite sequence of steps that restores coherence and thus ensures progress and completion. A session environment Δ is *initially coherent* if it is obtained from a coherent environment, i.e., $\Delta_0 \mapsto^* \Delta$ for some coherent Δ_0, and does not contain optional blocks without their counterparts.

Progress ensures that well-typed processes cannot get stuck unless their protocol requires them to. In comparison to standard formulations of progress, we add that the respective sequence of steps does not require any optional blocks to be unreliable. We define an optional block as *unreliable* w.r.t. to a sequence of steps if it does fail within this sequence and else as *reliant*. In other words we ensure progress despite arbitrary (and any number of) failures of optional blocks.

Completion is a special case of progress for processes without infinite recursions. It ensures that well-typed processes, without infinite recursion, follow their protocol and then terminate. Similarly to progress, we prove that completion holds despite arbitrary failures of optional blocks but does not require any optional block to be unreliable.

A simple but interesting consequence of this formulation of completion is, that for each well-typed process there is a sequence of steps that successfully resolves all optional blocks. This is because we type the content of optional blocks and because the type system ensures that these contents reach exactly one success reporting message $[\mathsf{r}]\langle\tilde{v}\rangle$ in exactly one of its parallel branches (and in each of its choice branches).

Theorem 1 (Properties).

Subject Reduction: *If $\Gamma \vdash P \triangleright \Delta$ and $P \longmapsto P'$ then there exists Δ' such that $\Gamma \vdash P' \triangleright \Delta'$ and $\Delta \mapsto^* \Delta'$.*

Progress: *If $\Gamma \vdash P \triangleright \Delta$ such that Δ is initially coherent, then either $P = \mathbf{0}$ or there exists P' such that $P \longmapsto^+ P'$, $\Gamma \vdash P' \triangleright \Delta'$, where $\Delta \mapsto^* \Delta'$ and Δ' is coherent, and $P \longmapsto^+ P'$ does not require any optional block to be unreliable.*

Completion: *If $\Gamma \vdash P \triangleright \Delta$ such that Δ is initially coherent and P does not contain infinite recursions, then $P \longmapsto^* \mathbf{0}$, $\Gamma \vdash \mathbf{0} \triangleright \emptyset$, and $P \longmapsto^* \mathbf{0}$ does not require any optional block to be unreliable.*

Reliance: *If $\Gamma \vdash P \triangleright \Delta$ such that Δ is initially coherent and P does not contain infinite recursions, then $P \longmapsto^* \mathbf{0}$ such that all optional blocks are successfully resolved in this sequence.*

The proofs of these properties can be found in [1]. They basically follow the same line of argument as used for similar type systems involving elaborate but straightforward structural inductions over the sets of rules.

Session types usually also ensure *communication safety*, i.e., freedom of communication error, and *session fidelity*, i.e., a well-typed process exactly follows the specification described by its global type. With optional blocks we lose these properties, because they model failures. As a consequence communications may fail and whole parts of the specified protocol in the global type might be skipped. In order to still provide some guarantees on the behaviour of well-typed processes, we however limited the effect of failures by encapsulation in optional blocks. It is trivial to see, that in the failure-free case, i.e., if no optional block fails, we inherit communication safety and session fidelity from the underlying session types in [2,3] and [7]. Even in the case of failing optional blocks, we inherit communication safety and session fidelity for the parts of protocols outside of optional blocks and the inner parts of successful optional blocks, since our extension ensures that all optional blocks that depend on a failure are doomed to fail and the remaining parts work as specified by the global type.

7 Conclusions

We extend standard session types with optional blocks with default values. Thereby, we obtain a type system for progress and completion/termination despite link failures that can be used to reason about fault-tolerant distributed algorithms. Our approach is limited with respect to two aspects: We only cover algorithms that (1) allow us to specify default values for all unreliable communication steps and (2) terminate despite arbitrary link failures. Accordingly, this approach is only a first step towards the analysis of distributed algorithms with session types. It shows however that it is possible to analyse distributed algorithms with session types and how the latter can solve the otherwise often complicated and elaborate task of proving termination. Note that, optional blocks can contain larger parts of protocols than a single communication step. Thus they may also allow for more complicated failure patterns than simple link failures/message loss.

We extend a simple type system with optional blocks. The (for many distributed algorithms interesting) concept of rounds is obtained instead by using the more complicated nested protocols (as defined in [7]) with optional blocks. Due to lack of space, the type systems with nested protocols/sub-sessions and optional blocks as well as more interesting examples with and without explicit (and of course overlapping) rounds are postponed to [1]. However the inclusion of sub-session is straightforward and does not require to change the above concept

of optional blocks. In combination with sub-sessions our attempt respects two important aspects of fault-tolerant distributed algorithms: (1) The modularity as e.g. present in the concept of rounds in many algorithms can be expressed naturally, and (2) the model respects the asynchronous nature of distributed systems such that messages are not necessarily delivered in the order they are sent and the rounds may overlap.

Our extension offers new possibilities for the analysis of distributed algorithms and widens the applicability of session types to unreliable network structures. We hope to inspire further work in particular to cover larger classes of algorithms and system failures.

References

1. M. Adameit, K. Peters, and U. Nestmann. Session Types for Link Failures (Technical Report). Technical report, TU Berlin, 2017. Available at `https://arxiv.org`.
2. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *Proc. of CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
3. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
4. S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):156–205, 2016.
5. M. Carbone, K. Honda, and N. Yoshida. Structured Interactional Exceptions in Session Types. In *Proc. of CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
6. R. Demangeon. Nested Protocols in Session Types. Personal communication about an extended version of [7] that is currently prepared by R. Demangeon., 2015.
7. R. Demangeon and K. Honda. Nested Protocols in Session Types. In *Proc. of CONCUR*, volume 7454 of *LNCS*, pages 272–286. Springer, 2012.
8. D. Kouzapas, R. Gutkovas, and S. J. Gay. Session Types for Broadcasting. In *Proc. of PLACES*, volume 155 of *EPTCS*, pages 25–31, 2014.
9. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
10. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I and II. *Information and Computation*, 100(1):1–77, 1992.
11. G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.