We thank the reviewers for their detailed comments.
We will incorporate their feedback in the next version of our paper: we will streamline the presentation of key concepts (dependencies, relative types, priorities) and new contributions (APCP, deadlock-freedom, routers); for this, the constructive feedback by Rev4/Rev3 is immediately actionable.

No technical issues were raised.
To keep our answer short, below we respond to selected points. For detailed responses to all points, see https://basvdheuvel.github.io/assets/static/ESOP21rebuttal.pdf
(The protocols by Rev3 look better in PDF.)

===Complex Algorithms===
Rev1 finds our approach overly complex, regarding
(a) amount of ingredients
(b) pseudocode presentation for projection (Fig2,p8) and synthesis (Fig4,p20).
We respectfully disagree, as we feel the criticism is not objective.

-Concerning (a):
Our approach does not rely on more notions than traditional MPST approaches.
We abandon local types/merge/subtyping, instead proposing relative types. Merge/subtyping add flexibility in protocols by *silently* expanding branches;
our approach uses dependencies, which make flexibility *explicit*.

-Concerning (b):
Our algorithms use pattern matching to distinguish six necessary cases.
In both algorithms, only some cases use nested ifs.
An algorithmic presentation of usual projection would follow a similar structure.
We realize that our pseudocode may obscure some key concepts;
we will incorporate the reviewers' feedback to address these presentational shortcomings.

===Proofs of Theorems 3&5===
Rev1 asks about these proofs.
They straightforwardly proceed by induction on G, following by construction.
We will add proof sketches.

===Modification to protocols===
Rev2-Rev3 are concerned about modifications to protocols.

We stress that our approach does not modify protocols.
Rather, relative projection adds synchronizations to ensure the protocol is implementable ("realizable").

Adding messages to ensure protocol implementability is a known approach.

Eg., Salaün et_al. use this approach in a similar context (see www.doi.org/10.1109/TSC.2011.9).
We discovered this paper after submission, and will add it as related work.

The reviewer indicates that adding messages restricts expressibility.
We disagree: in fact, it merely shifts expressibility.

===Protocols with Recursion===
Rev3 had trouble grasping our handling of recursion. They propose the following protocol; it allows us to explain our approach's ingredients, and how it can analyze protocols not covered by usual MPSTs.

```
    ⎧ e. μX. p→q ⎧ l. p→q { l. X }   ⎫         ⎫
p→r ⎨            ⎩ d. q→p { e. end } ⎭         ⎬
    ⎩ o. p→q ⎧ l. μX. q→q ⎧ l. p→q { l. X }  ⎫ ⎪
    ⎪        ⎪            ⎩ d. q→p { o. end} ⎭ ⎭
```

A message q→q (in the o,l branch) doesn't make sense; we assume the message q→p.

This protocol is not well-formed in traditional approaches:
eg., for the projection of the e-branch onto q, the input *from* p in the l-branch cannot be merged with the output *to* p in the d-branch.

We project the protocol onto p,q:

```
G | (p,q) = p!r { e. muX. p { l. p { l. X },
                             d. q { e. end } },
                  o. p { l. muX. q { l. p { l. X },
                                     d. q { o. end } } }
```

In both branches (e,o), relative projection conserves muX.
Now, relative projection onto p,r reveals a dependency between p,q:

```
G | (p,r) = p { e. muX. p!q { l. skip .X,
                             d. skip .end },
                o. skip. muX. p?q { l. skip. X,
                                    d. skip. end }}
```

Although the protocol specifies no direct exchanges between p and r after message p→r, both branches e and o contain a dependency (p!q, p?q) to decide whether to end or to loop.

In favor of consistency, our projection marks branches with differing projected choices as dependencies (here `skip.X` ≠ `skip.end`).
In projection, if a recursion body has no communications (ie., is equal to a sequence of `skips` followed by `end` or a recursion variable), then recursion is not preserved.
Hence, here the dependency (`p!q`, `p?q`) is crucial to preserve recursion.
Eg., in the e-branch, relative projection yields `muX.p?q { l. skip.X, d. skip.end }`.
Although we would like this projection to simply yield `end`, this kind of additional communications is redundant but harmless;
Our projection can be optimized to avoid these cases.

Rev3 asks about a similar second protocol, which we omit here due to space limitations (the only difference is the placement of recursion).

===Unclear/Complex Initial Sections===
Rev3 and Rev4 indicate roadblocks for readability, in particular in the initial sections. We will integrate their constructive feedback.

Regarding APCP, Rev4 asks:

> (p10): "Note that the finite fragment of APCP (without recursion) retains the strong Curry-Howard foundations" -> even without recursive types the fact that there can be circular structures (at least that's what I understood from the previous paragraph) seem to be compromising the Curry-Howard foundation. Can you comment? Also, I see that you have MIX, which to my understanding causes problems for cut-elimination.

APCP is based on PCP, which rests upon Curry-Howard foundations via a Classical Linear Logic with priorities.
PCP rules out circular dependencies exploiting priorities following type systems by Kobayashi.
APCP extends PCP with recursion. Our approach to deadlock-freedom is based directly on that for PCP.
By “finite fragment of APCP” we mean APCP without recursion (asynchronous PCP). Properties for this fragment rely on Curry-Howard foundations.
Support for recursion is an orthogonal enhancement, and does not rely on logical foundations.

> (Fig3/p14): Rule Cycle: It seems this rule gives rise to circular dependencies because the two endpoints are not in different sessions. This could give rise to a deadlock. How is such a rule sound?

Thm2 (§4) addresses this key point. We will clarify this.

> (Fig3/p14): Tensor rule: I would expect the continuation type z:B^bottom to be a premise. This seems rather nonstandard. What is the reason for it? [Reading on: apparently it is connected to priorities, but can you elaborate?]

An asynchronous output, typed by TENSOR, is an atomic action; there is no continuation process to serve as premise.
The continuation can be provided with MIX/CYCLE.
Rule TENSOR^\star in Fig3 (p13) combines TENSOR/MIX/CYCLE to provide a rule with the continuation as a premise.
The text just before Prop2 (p14) already provides explanations.

---------- REVIEW 1 ----------
SUBMISSION: 2
TITLE: A Decentralized Reduction of Multiparty Protocols to Binary Session Types
AUTHORS: Bas van den Heuvel and Jorge A. Pérez

---------- Overall evaluation ----------
SCORE: -1 (weak reject)
----- TEXT:
The paper presents a new scheme for projecting multiparty session types into binary session types, by introducting the notion of relative types.

I think I understand the objective of this work: address session interleaving, asynchronous communication, and recursion in the projection of global types into binary session types, concepts that other solutions in the literature find difficulties in addressing.

We would complement this summary by mentioning that another objective is to enable a decentralized implementation of multiparty session types.

I can see novelty in the paper. At the level of processes we have a new binary session language that is asynchronous and features for this purpose a) binary message exchanges (the message itself and the continuation and b) choice operators that explicitly exchange the continuation channel. And this variant of the calculus may unleash the intended extension sought by the paper.

Indeed, the APCP calculus is another contribution of the paper, of independent interest.

But the complexity of the solution is detrimental. One has global types (def 1) and relative types (def 2) and session types (def 7), so that we have two forms of local types.

We respectfully disagree.
Our solution based on
- global types, relative types and session types
is not more complex than previous works, which rely on
- global types, local types, merge, subtyping, and (binary) session types.

Then we need to translate relative types into session types. Two translations are proposed. There is a result (Proposition 4) relating one of the translations, but not the other.

Here the reviewer seems to be confused: Proposition 4 concerns both translations, as it relates session types obtained from relative projections of global types.

The projection function to relative types (fig 2) is too complex to be understood by a human mind. [...] It is 25 lines long. I cannot make sense of it. The language used to

describe the function does not help. A little pattern matching and definition by cases, as you find in all functional languages, would help. But the problem remains. There are are just too many (nested) conditionals. The case of message passing alone includes 8 (eight) ifs.

As in other proposals for multiparty session types, implementing a global behaviour by means of several binary processes uses brokers or message routers. These routers can be synthesised from types (fig 4) , but again the synthesis function is way too complex. [...] The synthesis of routers function is 27 extremely long lines. Its definition barely fits in a page. Again, I cannot make sense of it. It is just too complex. See comment on fig 2.

Here again complexity is debatable.
Our formulations already use pattern matching. Projection (Fig.2) and router synthesis (Fig.4) consist each of six independent case clauses (one per global construct), the most complex of which uses three nested ifs (for directed messages).
It appears the reviewer is confused by how the algorithms generate their output depending on the form of global type.

We want to prove results on these definitions. Given definitions of this sort, the chance that human-made proofs come out flawed is huge. Think of the proof of Thm 3 (whose proof is not shown, not even in the appendix). It must go by induction on G. For each case and subcase, a type derivation has to be written. Given def of synthesis (Fig 4), the number of cases and their complexity is overwhelming. Perhaps machine-aided proofs would help here.

The proof of Thm 3 is straightforward and follows by construction; we will add a proof sketch to indicate so.
Using machine-aided proofs is an excellent suggestion; working with typed pi-calculus processes in proof assistants is an exciting open problem.

Theorem 5 is another example of a result that involves synthesis, for which no proof is found. This result probably is even more challenging for it involves process equivalence in addition to a 27 lines long definition.

The proof of Thm 5 is also straightforward by construction.
Due to space constraints we were not able to present the related algorithm, let alone a proof with many, largely repetitive cases.

Minor comments.

The font used for type and processes in display mod is too small. The types and processes are complicated enough. Making them smaller can only exacerbate the problem.

We will increase the font size.

> The choice of examples. There all sorts of examples in the paper. No conducting line. For example, Example 1 (Page 9) is really example 2 and is different from the one in the introduction. The authors could select one (or more) examples to use throughout the paper.

We used several different examples in order to highlight several aspects of our approach. It is an easy fix to give the example in Sec. 2 a number.

> Page 4. The "projection of the this branch is skip.a{auth<bool>.X}". I suspect you mean G in place of X. A type if a free X cannot be well-formed.

There appears to be some confusion here. First, well-formedness is defined on global types, not on relative types. Second, the mentioned relative type is projected from a branch in G; the X in this branch is bound in the full projection of G: `μX.s!c{login.skip.a{auth<bool>.X}, quit.skip.end}`.

> Page 4. Italics boldface (relative types, router) is a bit too much and probably goes against typographical conventions.

We will change this to boldface only.

> Page 5. The bidirectional link between x any is written as a directional arrow from x to y. Why innovate? Caires and Pfenning (and probably older works by Honda) use the obvious bidirectional arrow <->.

We follow the notation by Dardha and Gay's PCP, where forwarding is presented with a directional arrow.

> Page 6, Def 1. "Message types include basic types (unit, bool, int) which are defined as end." What do you mean by this? That the grammar should read as below?
>
> S, T ::= end | unit | bool | int | !T.S | ...
>
> Then !int.bool would be a type, which is not what you want. If you want to add unit, bool, int, then you need two syntactic categories (or else remove the sentence).

We mean that basic types such as `unit` and `bool` are just closed channels, i.e. `unit` and `bool` are just equal to `end`. We will revise this explanation.

> Fig 2, line 20. Should it be "for some j in J"? How can one return a type for all j in J?

There is some confusion: the line says "for *any* j in J", which is equivalent to "for *some* j in J".

Fig 3, Rule Rec. This rule does not work with processes of the form "mu X(). my Y(). P".

This is not correct: Rule Rec in Fig.3 can indeed type processes of the mentioned form.

Example 1. Is 1 part of the language?

We assume the reviewer means Example 3.
In this case, **1** denotes a channel representing the number 1. We will improve this explanation.

---------- REVIEW 2 ----------
SUBMISSION: 2
TITLE: A Decentralized Reduction of Multiparty Protocols to Binary Session Types
AUTHORS: Bas van den Heuvel and Jorge A. Pérez

---------- Overall evaluation ----------
SCORE: 0 (borderline paper)
----- TEXT:
Summary

The authors approaches the topic of multiparty session types (MST) by reducing it to binary sessions and adding extra synchronization in the form of "router" processes.

MST use a global description of a communication protocol and then project the protocol on each participants. However, this approach has some limitations as not every communication protocol can be projected. In particular, projecting is not possible if a process does not learn about a choice that influences its behavior.

On the other hand, binary session types (BST) are better understood and have a cleaner theory. Therefore, the authors try to reduce MST to a collection of BST. In this work, the author project the protocols on binary session (interactions of two participants) rather than a single participants and do not completely erase the information about the other process. The projection keeps information about choices of other processes have some influence on the current interaction. This extra information is then used by router processes which send extra messages to guarantee the correct execution of the protocol.

The type system includes choice, parallel composition (where each part has disjoint set of participants), recursion, and delegation. The delegation, i.e., transfer of channels in messages, is specified in terms of a channel endpoint type.

There is no local types (projection on one participant) from traditional MST but relative types. Relative types are the projection on pairs of participants. Furthermore, if a choice involves only one of the two processes but has some influence, the relative type keeps this information.

To show the properties of the type system (Subject Reduction, Deadlock-Freedom), the authors develop the APCP calculus. APCP is based on linear logic and priorities. These two elements combine to create a proofs of a specific from from which the two properties follow.

What ties APCP and the relative projection are router processes which sits between the participants. The authors give a procedure that synthesize a router a process. A router looks at all the relative types in which a process is involved and synchronizes them. The

router adds extra messages to communicate to other routers the outcome of choices which impact a process but is not directly observed by it.

This is a very accurate summary.

Comments

MST and the study of how communication protocols can be safely implemented is an important topic and challenging topic. The introduction of router processes as a way of making choice visible is an interesting approach. However, it has some drawbacks.

Part of the fundamental question is what kind of alteration to the protocol is justifiable. If an MST specification is supposed to be implemented as is then the approach is not viable as it insert both processes and messages. If we allow changing the protocol then what change makes sense?

This seems to be a misunderstanding: as mentioned above, our approach does not alter protocols.

If we consider adding extra data to messages then the question has already been mostly solved. (Mostly as the models are a bit different.)

Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. J. Comput. Syst. Sci., 2006. https://doi.org/10.1016/j.jcss.2005.09.007

BTW, the authors should explain to me where in that work extra synchronization messages and centralized controller are added.

Extra synchronization messages are discussed in Sec. 5.2, on p. 26 (642).
We will rephrase the comment about the centralized controller, since we misunderstood this point.

The earlier work did not have much success and I attribute that to the fact that people are reluctant to modifying the protocol specification. Therefore, I'm not particularly enthusiastic about the approach. The overhead has not been evaluated on realistic protocols (in the worst case one message becomes n messages where n is the number of processes).

Our approach does not modify protocol specifications.

The technical developments is sections 4 and 5 is very difficult to follow for someone not familiar with the work on which APCP is built. The notations can at time be impenetrable.

We now realize that presentation can be improved to be self-contained wrt previous works. Specific pointers to impenetrable notation are appreciated.

> While I trust the technical development presented by the authors, I have some trouble to understand the choices they make for the model and what kind of protocols the MST they propose can handle.

We consider a syntax of global types that is entirely standard. We give multiple examples illustrating the class of protocols that can be analyzed with our approach.

> In particular part about the handling of the recursion I could not grasp. Maybe the authors could help me understand how their type system would work on the following two examples:

```
      ⎧ e. μX. p → q ⎧ l. p → q { l. X }   ⎫           ⎫
p → r ⎨              ⎩ d. q → p { e. end } ⎭           ⎬
      ⎪ o. p → q ⎧ l. μX. q → q ⎧ l. p → q { l. X }  ⎫ ⎪
      ⎩          ⎩              ⎩ d. q → p { o. end} ⎭ ⎭
```

A message q→q (in the o,l branch) doesn't make sense; we assume the message q→p.

This protocol is not well-formed in traditional approaches:
eg., for the projection of the e-branch onto q, the input *from* p in the l-branch cannot be merged with the output *to* p in the d-branch.

We project the protocol onto p,q:

```
G | (p,q) = p!r { e. muX. p { l. p { l. X },
                             d. q { e. end } },
                  o. p { l. muX. q { l. p { l. X },
                                     d. q { o. end } } }
```

In both branches (e,o), relative projection conserves muX.
Now, relative projection onto p,r reveals a dependency between p,q:

```
G | (p,r) = p { e. muX. p!q { l. skip .X,
                              d. skip .end },
                o. skip. muX. p?q { l. skip. X,
                                    d. skip. end }}
```

Although the protocol specifies no direct exchanges between p and r after message `p→r`, both branches e and o contain a dependency (`p!q`, `p?q`) to decide whether to end or to loop.
In favor of consistency, our projection marks branches with differing projected choices as dependencies (here `skip.X ≠ skip.end`).
In projection, if a recursion body has no communications (ie., is equal to a sequence of `skip`s followed by `end` or a recursion variable), then recursion is not preserved.
Hence, here the dependency (`p!q`, `p?q`) is crucial to preserve recursion.
Eg., in the e-branch, relative projection yields `muX.p?q { l. skip.X, d. skip.end }`.
Although we would like this projection to simply yield `end`, this kind of additional communications is redundant but harmless;
Our projection can be optimized to avoid these cases.

```
      ⎧ e. μX. p → q ⎧ l. p → q { l. p → q { l. X } } ⎫              ⎫
p → r ⎨              ⎩ d. q → p { e. end }            ⎭              ⎬
      ⎪ o. μX. q → q ⎧ l. p → q { l. p → q { l. p → q { l. X } } } ⎫ ⎪
      ⎩              ⎩ d. q → p { o. end }                          ⎭ ⎭
```

The examples should render properly with a monospace font.
For conciseness, the message payload are omitted (always `<unit>`).

This protocol is handled in a similar way as above; only the placement of the recursion differs.

---------- REVIEW 3 ----------
SUBMISSION: 2
TITLE: A Decentralized Reduction of Multiparty Protocols to Binary Session Types
AUTHORS: Bas van den Heuvel and Jorge A. Pérez

---------- Overall evaluation ----------
SCORE: 0 (borderline paper)
----- TEXT:
The paper as a whole looks promising, it tackles an interesting problem and potentially provides a solid technical solution for it. However, I had problems understanding the concepts presented in the paper. In particular the initial sections seems unnecessary complex. Further, I am not convinced by the solution presented as an alternative for merge (adding message, which in essence restricts the allowed branching pattern, in global types).

We understand and appreciate this criticism. We will address it by simplifying the introductory example (removing the recursion is a good suggestion).
This simplification should lead to extra space to better explain the concepts.

There seems to be confusion concerning the restrictiveness of our method. Many branching patterns that are not allowed by merge-based approaches are allowed in our approach, e.g.

```
p -> q {a<unit> .q -> r {b<unit> .end},
        c<unit> .r -> q {d<unit> .end}}.
```

Please see also our point in the general comments: our approach does not add messages in the sense of modifying the given global protocol.

Below further comments:

* IMO the way you structured the introduction you should also discuss "A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming" in it.

We will expand the current (brief) reference to this paper.

* Compact MPST examples are sometimes a bit contrived however the example in the beginning of 2 goes a bit too far.

We don't think the example is more contrived than the usual two-buyer-seller example.

* I find the explanation in sec 2 hard to follow.
  * The text fails to provide me with an intuition on what the concepts are. E.g. I am missing an intuition on relative types, as they are more than just describing the "peer-to-peer perspective between each pair of participants"
    - Potentially adding a sentence or two at the beginning on what "dependencies" are, would be helpful.

These suggestions can be easily incorporated.

* It uses technical knowledge only later defined, which are neither sufficiently/at all explained nor intuitive to me. E.g. I know MPST but can not parse "a{auth<bool>.X}" or why do you add "skip" (I can answer those after reading the paper but not the first time I read Sec 2).

We will add an explanation of the syntax of relative types in Sec. 2.
We use “skip” for technical reasons, which are irrelevant to the example at this point. We will revise this passage.

* IMO skip and the "combining" of relative types should be explained earlier.

We will explain “skip” earlier.
We are not sure what is meant by the “combining” of relative types,

* In my first pass I gave up on the router example, it is too complex. I would consider to at least remove recursion from the example and potentially to not have a router implementation at all and to just conceptually explain routers (while giving the formal example after you introduced the notation).

We will replace the technical router example with a more intuitive, conceptual explanation.

I don't see the advantage of the "different topologies". Is the rooter synthesis different in any of those cases?

No, and this is a key point: router synthesis is independent from the actual topology for interacting parties.
The advantage is that our approach is not limited to a centralized approach: we can model fully distributed implementations, but also centralized implementations.

* Sec 3:
The differences and similarities between relative types and local types should be explained more.

We will add more details on this (currently given in the appendix).

* I find the usage of "r" and "s" in "r -> s {..}", in projection, confusing. "s" signals a "send" or "sender" to me and "r" signals a "receive" or "recipient, but the usage is the other way around.

We never meant to assign any meaning to “r” and “s” as “recipient” and “sender”, respectively - we will fix this confusing notation.

* IMO using the "normal" style, used to write projection, for relative projection instead of the pseudo code would make Fig 2 easy to read. Further, the 3 level (or 4 if you also count the case) of conditions for the interaction case "r -> s" seem unnecessary.

Projection can be easily given as a coinductive definition. The several levels of conditions are necessary to distinguish the several patterns of communication in a concise way.

* The notation "p -> q {l<S>}.G" makes it easy to confuse a singleton interaction and a branching case (I needed two attempts to find the branching of the Two-Buyer-Seller example). I am personally used to dropping the "{}" in a singleton interaction.

This is easy to fix.

Example 1. G | (a,s) has a s?b{..} receive dependency with b but I don't see any send for that receive. Can you please elaborate on that.

The type s?b{..} denotes a dependency on s receiving a choice from b.
The send for that receive will appear in the relative projection onto s and b.
Note that we do explain the syntax of relative types below its definition (Def. 2, p. 7).

"Our approach induces explicit messages; in our view, this is an acceptable price to pay for a notion of well-formedness based exclusively on projection and not reliant on extraneous notions, such as merge and subtyping. It is easy to see that in a global type with n participants, the amount of messages per communication is O(n)"

Introducing messages is a big deal, using merge allows to extend the range of expressible branching patterns (whereas the added messages restrict the expressible branching pattern). In essence the added messages add something like an implicit multicast [1] to the global type and then your branching does not require merge. I.e., the added message adds synchronization to a MP interaction which is not needed when projections use merge.

[1] Coppo et al., Global Progress for Dynamically Interleaved Multiparty Sessions

The added synchronizations are indeed reminiscent of an implicit multicast.
However, note that dependencies multicast only to those participants that need the information, not to all participants.
Although merge allows to realize specific branching patterns, it requires additional definitions and proofs, and implicit flexibility in protocols.
On the other hand, adding synchronizations is an established method of making global protocols realizable.

We do not agree that adding messages restricts the range of expressible patterns; in fact, it merely supports a different range of expressible patterns.

Consider, e.g., the following example which is not supported by merge, but can be realized by adding synchronizations:

```
p -> q {a<unit> .q -> r {b<unit> .end},
        c<unit> .r -> q {d<unit> .end}}.
```

The way the paper deals with branching is ok with me however I am missing a more nuanced discussion on what the consequences are and I find the repeated highlighting that the work does not need merging a bit irritating.

Our intention was not to irritate the reader with our comparisons with merge; abandoning merge and subtyping is a distinguishing point that we will stress differently.
We will add a more nuanced discussion in the main text (currently given in the appendix).

Minor:
* Is APCP an acronym?

Yes, it stands for “Asynchronous Priority-based Classical Processes”

* I find the following sentences in Sec 3 difficult to parse.
  * "The relative type end indicates the end of the interaction between the type’s participants." (the "type’s participants" confused me the first time i read the sentence)
  *" Type p{` i hS i i.R i } i∈I specifies that p must choose a label li for some i ∈ I and send it to the other participant along with a message of type S" (should specify who the other participant is).

A relative type expresses the interactions between a pair of participants from a vantage point. As a result, an exchange only has to mention one of the participants: the recipient of a message is implicit. We will phrase this in a better way.

---------- REVIEW 4 ----------
SUBMISSION: 2
TITLE: A Decentralized Reduction of Multiparty Protocols to Binary Session Types
AUTHORS: Bas van den Heuvel and Jorge A. Pérez

---------- Overall evaluation ----------
SCORE: 0 (borderline paper)
----- TEXT:
SUMMARY

This paper contributes an extensive treatment of multiparty session types in terms of a new form of binary session types. The key abstraction necessary to bridge between global types and binary session types seems to be the notion of a relative type, a generalization of a local type that projects a global type onto two participants rather than one. The formalization of these new binary session types results in the language APCP, for which preservation and deadlock-freedom (based on the notion of a live process) hold. APCP supports asynchronous communication via explicit continuation channels, appealing to the distributed nature of multiparty session types, and deadlock-freedom through Kobayashi-style priorities. Moreover, APCP supports recursive types, all improvements over prior work. By means of an operational correspondence, the authors show that deadlock-freedom transfers from binary to multiparty session types. Proofs and further examples are provided in the appendix.

This is a very accurate summary.

EVALUATION

I gather that this submissions collects an extensive amount of solid work. Also, I agree with the authors that it is instructive and beneficial to clarify the connection between multiparty and binary session types. From these angles, I would be very much inclined to give this paper a higher score. However, I really had troubles understanding the work at a more detailed level, even though I am knowledgeable and I very carefully read the paper and spent a considerable amount of time on it. As a result I am not sure whether this paper is ready for publication without undergoing a considerable revision.

We will improve our explanations by incorporating the reviewer's useful feedback.

I am listing below comments that I wrote while reading the paper. As such they are a good reflection of what possible roadblocks a reader could encounter. But let me try to explain where I got stuck. My problems already started on page 4. Here, the important notion of a relative type is explained. But somehow up to now, I'm still not getting a clear and detailed grasp of it. What is missing to me is somehow a decomposition of ideas. What was your main intuition to introduce a relative type. Why are relative types

superior to local types? I gather it's the fact that they talk about two participants and possible dependencies from other participants, but somehow I don't have a clear understanding. Then, I got totally confused by routers and implementations, how they relate to each other and how the relate to relative types. And somehow all these little uncertainties accumulated…

The main intuition behind relative types is that they describe the interactions between two participants; this allows us to decompose a global protocol into many peer-to-peer protocols. Working with peer-to-peer protocols is more intuitive than local protocols when defining a decentralized implementation of the given global protocol.

Relative types are not tied to dependencies; relative projection can be defined using the usual merge operator.
Such a definition requires adding subtyping for proofs of typability.
To reduce the number of auxiliary notions, we opted for dependencies, which allow for an expressive global protocol language.

COMMENTS

(p.2): "to define how matching actions for a local type are scattered across the local types of other participants" -> maybe better "in which relative order the various actions of local types must happen."

Thank you for the suggestion.

(p.2): recursion -> pls clarify whether term-level or type-level recursion.

We mean recursion in global types; we will clarify this in the text.

(p.4): The writing of the paragraph starting with "To analyze G we first obtain the peer-to-peer perspectives" must be improved, it is not clear. Especially the part "is a dependency for" is not crisp. I gather what is done is to look at the global type from the point of view of two endpoints and distill for each action whether one of these endpoints is involved, in which case the corresponding action is extracted, or not, in which case a skip action is extracted. If that's right, that's more or less what should be said.

Thank you for pointing out that this is not clear, we will improve it.

Reading on, looking at the first projection, I realize that my interpretation is wrong. So I now wonder: why is the exchange between c and a not a dependency for the exchange between s and a?

Your initial interpretation is in fact correct.

In this case, the exchange between c and a is not a dependency for the exchange between a and s, because there is only one branch.
Hence, a and s do not need to know which branch c and a have taken, as the result is always the same from their perspective.

> What do you mean by "The projections of the branches are not equal, so the initial exchange in G between s and c is indeed a dependency for the subsequent protocol between s and a."?

The initial exchange between s and c has two branches `login` and `quit`.
The projection of the `login` branch onto s and a is
`skip.a{auth<bool>.X}`,
while the projection of the `quit` branch onto s and a is
`skip.end`.
Since these projections are not equal, the initial exchange between s and c is a dependency for the interactions between s and a.
Note that we do explain this in the beginning of the mentioned paragraph.

> (p.5) "The router Rc is meant to communicate with its (unspecified) implementation" -> I thought that a router is an actual implementation? What then is an implementation?
>
> Can you explain what a difference between a router and an implementation is? As far as I can tell, a router is on the term-level as it follows your process syntax just introduced at the bottom of page 4 and top of page 5. Figure 1 seems to suggest that they are different entities.

Routers and implementations are indeed both typed APCP processes.
Routers rely on an implementation to provide and consume choices and values in order to execute a protocol, i.e. routers abstract away from specific implementation.
We will clarify this in the text.

> (Fig.1): This figure isn't explained in detail. For example, what is the semantics of the rectangles and hexagons? All I can see is that in all cases the connection lines (which must be channels) between the routers and implementations are the same. So I'm not sure in which way left is decentralized, middle centralized, and right hybrid.
>
> "Boxes mark connections (inner first)" -> I am not sure what that really means.

The different shapes do not have a specific meaning.
The idea of the figure is that outlines indicate connections of channel endpoints (parallel composition + restriction) using a sequence of MIX followed by a sequence of CYCLE.
The outlines that are inside represent channel restrictions that happen inside other restrictions.
We will improve the explanation of this figure.

To clarify:
- In the left figure, the routers are each first connected to implementations, and then to each other.
In other words, we first "hide" the communications between router and implementations, and then connect the several distributed participants.
- In the center figure, the routers are first connected to each other, and then to the implementations.
In other words, we first "hide" the communications between the routers, effectively creating a centralized mediator process, to which we connect the several implementations.

> (p.6): "Message exchange is asynchronous: a session can continue before a prior message has been received" -> although I understand what is meant, it would be better reformulated as something along the lines "a sender can continue with the session regardless of whether the sent message has already been received"

Thank you for the suggestion.

> (Fig.2): Is there a way to define the relative projection coinductively over the structure of G? At least the recursiveness of the algorithm suggests that, superficially.

We opted for an algorithmic presentation because we plan to implement the work in this paper. Following this and the other reviews, we will give a coinductive definition.

> (p.9): Discussion of Fig.2: could you provide an intuition for the treatment of exchanges?

If both p and q are involved (i.e. they're sender and recipient), the exchange simply becomes an exchange in the relative type.
If only one of p and q is involved, the exchange can become a synchronization message if necessary --- when the projections of the branches are not equal. If the exchange can be skipped --- when the projections of the branches are equal --- it will be.
If not all projections of the branches are equal, but p and q are not involved, the projection is not defined.
Please note that we do give an intuition to how the algorithm works, following Proposition 1 (p. 8).

> (p.10): "by moving from analyses based on purely centralized network topologies to a topology-agnostic analysis" -> can you elaborate? also, I don't understand the word "analysis" is this context.

In this context, "analysis" means modelling global protocols as distributed processes and establishing key correctness properties, in particular deadlock-freedom.
Our process calculus, APCP, allows us to do so.

However, by connecting processes in different orders, it is also possible to model global protocols in, e.g., centralized topologies.
Hence, we generalize over existing such analyses, which exclusively rely on centralization.
We will clarify our use of the word “analysis” in the text.

(p.11): "there are no empty outputs and input in APCP for closing channels explicitly" -> up to this moment I kept wondering whether channels are treated linearly or not. At least it seems to be the case that weakening is permitted. But then, how does that tie in with the earlier statement about the Curry-Howard correspondence?

All channels are linear.
The “weakening” you refer to concerns only closed channels, i.e. those of type \bullet.
Thanks for pointing this out, we will add a comment about it.

(p.11): Is the name y in output free or bound?

All channels in the output atom are free.
We will clarify this in the explanation below the grammar of APCP processes.

(p.12): priority -> it would be helpful to provide some intuition (for readers not familiar with prior work). And on the next page, it is said that unfolding increased the priority. Again, it would be helpful to provide some intuition. [Reading on: the explanation is at the bottom of p.13, which now also explains why unfolding increases priority. Some intuition should be provided right from the get go.]

We will clarify these points.

(p.13): For the typing judgment, is Gamma an affine context?

\Gamma is a linear context; we will mention this.

(related work): comparison with work on deadlock-freedom is very superficial.

The introduction and Sec. 4 extensively discuss deadlock-freedom, central to our approach.
We will add a more in-depth discussion.

QUESTIONS

(p.10): "Note that the finite fragment of APCP (without recursion) retains the strong Curry-Howard foundations" -> even without recursive types the fact that there can be circular structures (at least that's what I understood from the previous paragraph) seem to be compromising the Curry-Howard foundation. Can you comment on this? Also, reading on, I see that you have MIX, which to my understanding causes problems for cut elimination.

APCP is based on PCP, which rests upon Curry-Howard foundations via a Classical Linear Logic with priorities.
PCP rules out circular dependencies exploiting priorities following type systems by Kobayashi.
APCP extends PCP with recursion. Our approach to deadlock-freedom is based directly on that for PCP.
By “finite fragment of APCP” we mean APCP without recursion (asynchronous PCP). Properties for this fragment rely on Curry-Howard foundations.
Support for recursion is an orthogonal enhancement, and does not rely on logical foundations.

> (Fig.3/p.14): Rule Cycle: It seems this rule gives rise to circular dependencies because the two endpoints are not in different sessions. This could give rise to a deadlock. How is such a rule sound?

Thm2 (§4) addresses this key point. We will clarify this.

> (Fig.3/p.14): Tensor rule: I would expect the continuation type z:B^bottom to be a premise. This seems rather nonstandard. What is the reason for it? [Reading on: apparently it is connected to priorities, but can you elaborate?]

An asynchronous output, typed by TENSOR, is an atomic action; there is no continuation process to serve as premise.
The continuation can be provided with MIX/CYCLE.
Rule TENSOR^\star in Fig3 (p13) combines TENSOR/MIX/CYCLE to provide a rule with the continuation as a premise.
The text just before Prop2 (p14) already provides explanations.

> ---