Sprout: A Verifier for Symbolic Multiparty Protocols

Elaine Li[1](✉), Felix Stutz[2], Thomas Wies[1], and Damien Zufferey[3]*

[1] New York University, New York, USA
efl9013@nyu.edu, wies@cs.nyu.edu
[2] University of Luxembourg, Esch-sur-Alzette, Luxembourg
felix.stutz@uni.lu
[3] NVIDIA, Zürich, Switzerland
rilaak@gmail.com

Abstract. We present Sprout, the first sound and complete implementability checker for symbolic multiparty protocols. Sprout supports protocols with dependent refinements on message values, loop memory, and multiparty communication with generalized, sender-driven choice. Sprout checks implementability via an optimized, sound and complete reduction to the fixpoint logic μCLP, and uses MuVal as a backend solver for μCLP instances. We evaluate Sprout on an extended benchmark suite of implementable and non-implementable examples, and show that Sprout outperforms its competititors in terms of expressivity and precision, and provides competitive runtime performance. Sprout additionally provides support for verifying custom functional correctness properties beyond implementability.

1 Introduction

Implementability is the decision problem at the heart of top-down approaches to protocol verification, including choreographic programming [6,12,13] , high-level message sequence charts [1,2,7–10,18–22] and session types [3,4,14,16,23,26]. Implementability asks whether a global protocol, specifying message exchanges between all participants from a bird's-eye view, admits an asynchronous distributed implementation, namely one that is deadlock-free and exhibits the same behavior as the global specification.

In [17], we identify a sound and complete characterization of implementability for *global communicating labeled transition systems (GCLTSs)*. GCLTS is an expressive semantic model of protocols that subsumes many existing fragments of multiparty session types [3,4,23,25,26] and choreography automata [11]. The characterization of GCLTS implementability consists of three *Coherence Conditions*: Send Coherence, Receive Coherence, and No Mixed Choice, which reduce implementability to reachability and co-reachability in the GCLTS. In a nutshell, these are 2-hyperproperties stating that from two locally indistinguishable global protocol states, a participant can either perform a send action that is enabled

*Damien Zufferey was working at SonarSource in Switzerland when this work began.

© The Author(s) 2025
R. Piskac and Z. Rakamarić (Eds.): CAV 2025, LNCS 15933, pp. 304–317, 2025.
https://doi.org/10.1007/978-3-031-98682-6_16

in both states, or perform a receive action that uniquely distinguishes the two states, but cannot choose between performing a send or receive action. *Symbolic protocols* finitely represent infinite-state protocols using dependent refinements and mutable register variables. [17] derives sound and complete Symbolic Coherence Conditions for GCLTS-eligible symbolic protocols, expressed as μCLP instances. μCLP [24] is a fixpoint logic featuring recursive predicates with least and greatest fixpoint semantics, where the predicate body is constrained by a first-order logic formula over a background theory.

In this paper, we present Sprout[§], the first sound and complete implementability checker for symbolic, multiparty protocols. Sprout takes as input a symbolic protocol, and first checks whether the protocol is GCLTS-eligible. If so, it proceeds to generate μCLP instances corresponding to the Symbolic Coherence Conditions from [17], which it then discharges to the μCLP solver MuVal [24]. If all instances return invalid, Sprout reports that the protocol is implementable; if one instance returns valid, Sprout reports non-implementable along with the specific states and transitions that violate implementability; otherwise Sprout returns inconclusive. Sprout is sound and complete relative to the completeness and soundness of MuVal.

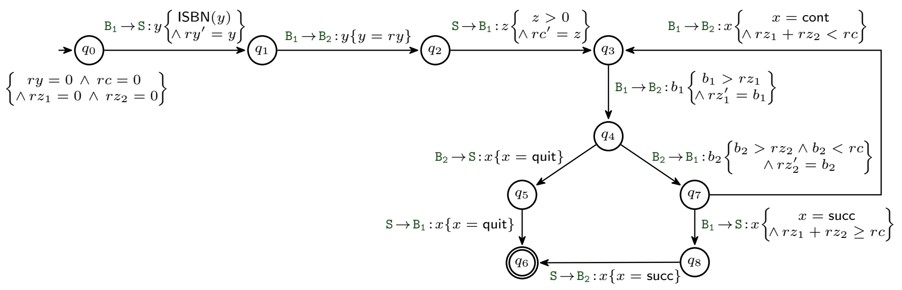
Sprout extends [17] with explicit GCLTS checking, optimized μCLP encodings of the Symbolic Coherence Conditions, and support for verification of functional correctness properties beyond implementability. We evaluate Sprout's expressivity, precision and efficiency against comparable tools [25, 26] on an expanded benchmark suite containing both implementable and non-implementable examples. Sprout is able to correctly classify protocols that are out of reach of its competitors, outperforming them in terms of expressivity and precision. In terms of efficiency, Sprout's performance is competitive. On multiparty protocols, its verification times vary with the size of the protocol and are largely bottlenecked by the efficiency of MuVal, although remaining in the order of seconds in most cases. We envision Sprout as a complementary intermediate step in existing top-down code generation toolchains for multiparty protocols whose implementability checks are incomplete.

2 Overview

2.1 Running Example

We introduce Sprout using the running example of the two-bidder protocol [17]. The two-bidder protocol specifies the message-passing behaviors of a seller `S` and two bidders $\texttt{B}_1$ and $\texttt{B}_2$, who negotiate to split the cost of a book. Bidder $\texttt{B}_1$ initiates the protocol by announcing a book title, identified by its ISBN number. Seller `S` informs $\texttt{B}_1$ of the book's price c, which is undisclosed to $\texttt{B}_2$. Then, $\texttt{B}_1$ and $\texttt{B}_2$ enter a bidding loop to determine their respective contributions b_1 and b_2. After $\texttt{B}_1$ proposes its contribution b_1, $\texttt{B}_2$ can either respond with a bid, or stop bidding by sending a quit message to `S`, who forwards it to $\texttt{B}_1$. The bidding continues until either $\texttt{B}_2$ chooses to stop, or the sum of the two bids exceeds c, in

[§] https://github.com/nyu-acsys/sprout



$\mathtt{B_1} \to \mathtt{S} : y \{\mathsf{ISBN}(y) \wedge ry' = y\}$
$\mathtt{B_1} \to \mathtt{B_2} : y\{y = ry\}$
$\mathtt{S} \to \mathtt{B_1} : z\{z > 0 \wedge rc' = z\}$
$\mathtt{B_1} \to \mathtt{B_2} : x\{x = \mathsf{cont} \wedge rz_1 + rz_2 < rc\}$
$\{ry = 0 \wedge rc = 0 \wedge rz_1 = 0 \wedge rz_2 = 0\}$
$\mathtt{B_1} \to \mathtt{B_2} : b_1\{b_1 > rz_1 \wedge rz_1' = b_1\}$
$\mathtt{B_2} \to \mathtt{S} : x\{x = \mathsf{quit}\}$
$\mathtt{B_2} \to \mathtt{B_1} : b_2\{b_2 > rz_2 \wedge b_2 < rc \wedge rz_2' = b_2\}$
$\mathtt{S} \to \mathtt{B_1} : x\{x = \mathsf{quit}\}$
$\mathtt{B_1} \to \mathtt{S} : x\{x = \mathsf{succ} \wedge rz_1 + rz_2 \geq rc\}$
$\mathtt{S} \to \mathtt{B_2} : x\{x = \mathsf{succ}\}$
q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8

Fig. 1: Candidate specification for the two-bidder protocol.

which case $\mathtt{B_1}$ informs $\mathtt{S}$ that the negotiation was successful. Refinements enforce that y is a valid ISBN number, and that $\mathtt{B_1}$ and $\mathtt{B_2}$'s bids are increasing from one round to the next, thus guaranteeing termination. A candidate specification for the two-bidder protocol is depicted in Fig. 1.

Sprout's input format closely follows the definition of *symbolic protocols*, formally defined over a set of participants $\mathcal{P}$ as follows:

Definition 2.1 (Symbolic protocol [17]). *A symbolic protocol is a tuple* $\mathbb{S} = (S, R, \Delta, s_0, \rho_0, F)$ *where*

- *S is a finite set of control states,*
- *R is a finite set of register variables,*
- *$\Delta \subseteq S \times \mathcal{P} \times X \times \mathcal{P} \times \mathcal{F} \times S$ is a finite set that consists of symbolic transitions of the form $s \xrightarrow{\mathtt{p} \to \mathtt{q} : x\{\varphi\}} s'$, where the formula φ with free variables $R \uplus R' \uplus \{x\}$ expresses a transition constraint over the old and new register values (R and R') and the sent value x,*
- *$s_0 \in S$ is the initial control state,*
- *$\rho_0 \colon R \to \mathcal{V}$ is the initial register assignment, and*
- *$F \subseteq S$ is a set of final states.*

The definition assumes a fixed but unspecified first-order background theory of message values (e.g. linear integer arithmetic). We denote by $\mathcal{F}$ the set of first-order formulas with free variables drawn from some set X that are interpreted over the set of message values $\mathcal{V}$. We assume standard syntax and semantics for first-order formulas. For a valuation $\rho \in X \to \mathcal{V}$ and $\varphi \in \mathcal{F}(X)$, we write $\rho \models \phi$ to indicate that φ evaluates to true under ρ in the underlying theory. Transition constraints simultaneously describe the current value being communicated and internal register updates. For $q_3 \xrightarrow{\mathtt{B_1} \to \mathtt{B_2} : b_1\{b_1 > rz_1 \wedge rz_1' = b_1\}} q_4$, the transition constraint both enforces that the bid b_1 sent from $\mathtt{B_1}$ to $\mathtt{B_2}$ is greater than $\mathtt{B_1}$'s previous bid, and describes a register assignment $z_1 := b_1$. For readability, we assume implicit equality constraints over unmentioned post-state registers that are not updated. The input file for our candidate specification is given in Fig. 2.

2.2 Implementability

Before checking implementability, Sprout first determines GCLTS eligibility. GCLTSs satisfy four assumptions: sink-finality, sender-driven choice, determin-

```
Initial state: (0)
Initial register assignments: ry=0, rc=0, rz1=0, rz2=0
(0) B1->S:y{(y>987000000000/\y<9880000000000)/\ry'=y} (1)
(1) B1->B2:y{y=ry} (2)
(2) S->B1:z{z>0/\rc'=z} (3)
(3) B1->B2:b1{b1>rz1/\rz1'=b1} (4)
(4) B2->S:quit{quit=0} (5)
(5) S->B1:quit{quit=0} (6)
(4) B2->B1:b2{b2>rz2/\b2<rc/\rz2'=b2} (7)
(7) B1->S:succ{succ=1/\rz1+rz2>=rc} (8)
(8) S->B2:succ{succ=1} (6)
(7) B1->B2:cont{cont=2/\rz1+rz2<rc} (3)
Final states: (6)
```

Fig. 2: SPROUT input file for protocol specification in Fig. 1.

ism, and deadlock freedom. Sink-finality states that only non-final states have outgoing transitions, sender-driven choice states that all outgoing transitions from the same state have a unique sender, determinism states that no transition can lead to two distinct post-states, and deadlock freedom states that every protocol run can be extended to a maximal run.

After confirming that our protocol is GCLTS-eligible, SPROUT proceeds to generate μCLP instances corresponding to the implementability characterization from [17], which consists of three Symbolic Coherence Conditions: Symbolic Send Coherence, Symbolic Receive Coherence and Symbolic No Mixed Choice. SPROUT generates the queries in negation form, and discharges them to the μCLP solver MUVAL [24]. SPROUT reports implementable if and only if all instances return invalid, indicating that all conditions are satisfied.

Unfortunately, SPROUT reports a violation to Symbolic Send Coherence for $\texttt{B}_2$ and the transition: `(4) B2->B1:b2{b2>rz2/\b2<rc/\rz2'=b2} (7)`. The violation indicates the existence of two global protocol states both with control state q_4 that are indistinguishable from $\texttt{B}_2$'s point of view, and a message value, such that sending the value to $\texttt{B}_1$ follows the protocol in one case but violates the protocol in the other. Closer inspection of this transition's constraint reveals that $\texttt{B}_2$ is required to send a bid that is strictly less than the price of the book c. However, c is not disclosed to $\texttt{B}_2$ during the protocol: $\texttt{B}_2$ is bidding in the dark. Thus, depending on the initial exchanges between $\texttt{B}_1$ and $\texttt{S}$, which are not observable to $\texttt{B}_2$, a bid could either satisfy or violate the middle conjunct, subsequently following or violating the entire protocol.

We can repair our candidate protocol by either omitting `b2<rc` from the aforementioned transition constraint, or by including a transition informing $\texttt{B}_2$ of the book's price before the bidding loop begins. Upon incorporating either fix, we find that all instances now return invalid as expected, and SPROUT reports that the repaired two-bidder protocol is implementable in $\sim$19s.

SPROUT also provides support for the verification of functional correctness properties beyond implementability. For example, we can verify that the sum of $\texttt{B}_1$ and $\texttt{B}_2$'s bids never decreases once they enter the bidding loop. This verification problem can be expressed in negation form as a μCLP instance as follows, where `stcon` is a least fixpoint predicate describing st-connectivity between two states in the global protocol:

```
exists  (s1: int) (ry1: int) (rc1: int) (rza1: int) (rzb1: int)
(s2: int) (ry2: int) (rc2: int) (rza2: int) (rzb2: int).
s1 > 3 /\
s2 > 3 /\
stcon s1 ry1 rc1 rza1 rzb1 s2 ry2 rc2 rza2 rzb2 /\
rza2 + rzb2 < rza1 + rzb1
s.t.
stcon (s1: int) ... : bool =mu
...
```

Sprout provides a suite of least and greatest fixpoint predicate definitions for defining custom verification queries that are then discharged to MuVal. MuVal confirms that this instance is indeed invalid in ∼9s.

3 Implementation

Sprout is implemented in ∼3500 lines of OCaml code. The tool and benchmarks used in the evaluation are available as part of the artifact accompanying this paper [15]. In this section, we describe aspects of its implementation, focusing on differences from the theory.

3.1 GCLTS Eligibility

The Coherence Conditions from [17] are precise for the GCLTS fragment of symbolic protocols, namely protocols that satisfy sink-finality, sender-driven choice, determinism and deadlock-freedom. Sink-finality and sender-driven choice are syntactic conditions that can be checked on the input protocol straightforwardly, without invoking MuVal. Determinism and deadlock freedom are undecidable in general. Sprout encodes the latter two as μCLP instances and discharges them to MuVal. We present the formal definition and μCLP encoding of each property below, assuming a symbolic protocol $\mathbb{S} = (S, R, \Delta, s_0, \rho_0, F)$ in the remainder of the section.

Determinism states that from a reachable protocol state, no transition can simultaneously satisfy two transition constraints that lead to two distinct post-states. Reachability is expressed as a least fixpoint in μCLP as follows:

Definition 3.1 (Reachability in symbolic protocol). *Let $s \in S$. Then,*

$$\mathsf{reach}(s', \boldsymbol{r'}) :=_\mu \quad (\, s' = s_0 \wedge \boldsymbol{r'} = \rho_0 \,) \;\vee\; (\bigvee_{(s,\, \mathtt{p}\to\mathtt{q}:x\{\varphi\},\, s') \in \Delta} \exists x\ \boldsymbol{r}.\, \mathsf{reach}(s, r) \wedge \varphi \,) \;.$$

The reach predicate takes as its arguments a control state s' and a set of registers $\boldsymbol{r'}$, which together constitute a symbolic protocol state. The first disjunct covers the base case in which s' is the initial state, and $\boldsymbol{r'}$ satisfy the initial register assignments. The second disjunct ranges over all transitions with s' as the post-state, and represents following a transition to reach s', which requires the transition predicate φ to hold in addition to *reach* on the pre-state s.

Equipped with the predicate reach, determinism is defined as follows.

Definition 3.2 (Determinism of symbolic protocol). $\mathbb{S}$ *is deterministic when for each pair of transitions* $s \xrightarrow{\mathsf{p}\to\mathsf{q}:x_1\{\varphi_1\}} s_1, s \xrightarrow{\mathsf{p}\to\mathsf{q}:x_2\{\varphi_2\}} s_2 \in \Delta$*, the following is valid:*

$$\forall x\ \boldsymbol{r}\ \boldsymbol{r_1'}\ \boldsymbol{r_2'}.\ \mathsf{reach}(s, \boldsymbol{r}) \wedge \varphi_1[x/x_1, \boldsymbol{r_1'}/\boldsymbol{r'}] \wedge \varphi_2[x/x_2, \boldsymbol{r_2'}/\boldsymbol{r'}] \implies s_1 = s_2 \wedge \boldsymbol{r_1'} = \boldsymbol{r_2'}\ .$$

Deadlock freedom states that every run in the protocol can be extended to a maximal run, meaning that it is either infinite or ends in a final state. Equivalently, we require that every reachable protocol state has an enabled outgoing transition, stated as follows.

Definition 3.3 (Deadlock freedom of symbolic protocol). $\mathbb{S}$ *is deadlock-free when for each non-final state* $s \in S \setminus F$*, the following is valid:*

$$\forall \boldsymbol{r}.\ \mathsf{reach}(s, \boldsymbol{r}) \implies \bigvee_{(s,\, \mathsf{p}\to\mathsf{q}:x\{\varphi\},\, s') \in \Delta} \exists x.\ \varphi\ .$$

For determinism, Sprout generates one μCLP query per state; for deadlock freedom, Sprout generates one μCLP query per pair of transitions sharing a pre-state. If the input protocol is not GCLTS-eligible, Sprout reports specifically which assumption is violated by which state or transitions.

The GCLTS checking step of Sprout is sound and relatively complete with respect to the completeness of MuVal, and Sprout only checks implementability of GCLTS-eligible protocols.

3.2 Optimizations

The first Sprout optimization elides implementability checking for binary protocols, which by [17, Lemma 5.10] are implementable by construction if they satisfy the GCLTS assumptions. Between checking GCLTS eligibility and generating implementability μCLP instances, Sprout checks whether the input protocol is binary, and if so, returns implementable immediately. Given that a large subset of benchmarks in the multiparty protocol literature are binary protocols, this optimization allows us to achieve performance within the same order of magnitude as existing tools for binary protocols, as we detail in §4.

The second and primary Sprout optimization concerns the encoding of Symbolic Coherence Conditions into μCLP instances. The conditions universally quantify over participants in the protocol, and then universally quantify over pairs of simultaneously reachable protocol states from the perspective of that participant. Together, the conditions rely on three recursive predicates: $\mathsf{prodreach}_{\mathsf{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2})$, which expresses that symbolic protocol states $(s_1, \boldsymbol{r_1})$ and $(s_2, \boldsymbol{r_2})$ are simultaneously reachable for p, $\mathsf{unreach}^{\varepsilon}_{\mathsf{p},\mathsf{q}}(s_2, \boldsymbol{r_2}, x_1)$, which expresses that p cannot follow ε transitions from $(s_2, \boldsymbol{r_2})$ to a state where it can send x_1 to q, and $\mathsf{avail}_{\mathsf{p},\mathsf{q},\mathcal{B}}(x_1, s_2, \boldsymbol{r_2})$, which expresses that message x_1 from p can be received by q from state $(s_2, \boldsymbol{r_2})$ without participants from $\mathcal{B}$ sending or receiving messages.

The Symbolic Coherence Conditions are defined as follows:

Definition 3.4 (Symbolic Coherence Conditions [17]). *Let* $\mathbb{S}$ *be a symbolic protocol. Then,*

- $\mathbb{S}$ *satisfies Symbolic Send Coherence when for each participant* $\mathtt{p}$*, transition* $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ *and state* $s_2 \in S$*:*

$$\mathsf{prodreach}_{\mathtt{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) \ \wedge\ \varphi_1 \wedge \mathsf{unreach}^{\varepsilon}_{\mathtt{p},\mathtt{q}}(s_2, \boldsymbol{r_2}, x_1) \implies \bot\ .$$

- $\mathbb{S}$ *satisfies Symbolic Receive Coherence when for every pair of transitions* $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ *and* $s_2 \xrightarrow{\mathtt{r}\to\mathtt{q}:x_2\{\varphi_2\}} s_2' \in \Delta_2$ *with* $\mathtt{p} \neq \mathtt{r}$*:*

$$\mathsf{prodreach}_{\mathtt{q}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) \ \wedge\ \varphi_1 \ \wedge\ \varphi_2 \ \wedge\ \mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(x_1, s_2', \boldsymbol{r_2'}) \implies \bot\ .$$

- $\mathbb{S}$ *satisfies Symbolic No Mixed Choice when for every pair of transitions* $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ *and* $s_2 \xrightarrow{\mathtt{r}\to\mathtt{p}:x_2\{\varphi_2\}} s_2' \in \Delta_2$*:*

$$\mathsf{prodreach}_{\mathtt{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) \ \wedge\ \varphi_1 \ \wedge\ \varphi_2 \implies \bot\ .$$

A μCLP instance is a pair $(\phi, \mathcal{R})$ of a *query* ϕ, which is a first order formula over a background theory, and a *body* $\mathcal{R}$, which is a sequence of inductive predicates with least or greatest fixpoint semantics. Symbolic Send Coherence in negation form thus naturally corresponds to one μCLP instance per participant. Each instance's query existentially quantifies over control states and registers, and is a series of $|Q| * |Q|$ disjuncts that perform case analysis over pairs of control states, i.e. each disjunct is of the form

$$s_1 = q_1 \wedge s_2 = q_2 \wedge \mathsf{prodreach}_{\mathtt{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) \ \wedge\ \varphi_1 \wedge \mathsf{unreach}^{\varepsilon}_{\mathtt{p},\mathtt{q}}(s_2, \boldsymbol{r_2}, x_1)$$

where $q_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:x_1\{\varphi_1\}} q_2 \in \Delta$. Each instance's body comprises the inductive predicates $\mathsf{prodreach}$ and $\mathsf{unreach}$, defined as least and greatest fixpoints respectively:

$$\mathsf{prodreach}_{\mathtt{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) =_{\mu} \ldots; \qquad\qquad \mathsf{unreach}^{\varepsilon}_{\mathtt{p},\mathtt{q}}(s_2, \boldsymbol{r_2}, x_1) =_{\nu} \ldots;$$

This naive encoding of [17]'s three conditions amounts to $3 * |\mathcal{P}|$ μCLP instances per protocol, where each instance is orders of magnitude larger than the average benchmark in MuVal's benchmark suite[¶], and the verification time for e.g. our running example exceeds 10 minutes. Thus, Sprout takes a different approach to structuring the Symbolic Coherence Conditions as μCLP instances. First, Sprout distributes each disjunct into a separate instance, yielding $|\mathcal{P}| * |Q| * |Q|$ instances for each condition. Next, Sprout decomposes the $\mathsf{prodreach}$ and $\mathsf{unreach}$ predicates by "currying" state arguments, generating one $\mathsf{prodreach}$ predicate per participant per pair of states, amounting to $|\mathcal{P}| * |Q| * |Q|$ predicate definitions, and one $\mathsf{unreach}$ predicate per pair of participants per state, amounting to $|\mathcal{P}| * |\mathcal{P}| * |Q|$ predicate definitions. We show in §4 that decomposing large instances into multiple instances with smaller queries and more inductive predicates improves the running time of MuVal by over two orders of magnitude for most protocols.

Thirdly, Sprout implements an overapproximation of simultaneous reachability that pre-filters pairs of control states before generating μCLP instances.

[¶] https://github.com/hiroshi-unno/coar/tree/main/benchmarks/muCLP/popl2023mod

Approximate simultaneous reachability disregards message values, only considering the sender and receiver of each event in a trace, e.g. $\mathtt{p!q}{:}4 \cdot \mathtt{r?p}{:}7 \cdot \mathtt{s!q}{:}5$ is abstracted to $\mathtt{p!q}{:}\text{-} \cdot \mathtt{r?p}{:}\text{-} \cdot \mathtt{s!q}{:}\text{-}$. This optimization preserves soundness and completeness of the tool: if two states are not approximately simultaneously reachable, then the Coherence Conditions say nothing about them; if two states are approximately simultaneously reachable, then the corresponding instances will be generated and checked, and in the case that they are not actually simultaneously reachable, will simply return invalid due to the prodreach conjunct being false.

Finally, for Send Coherence instances concerning simultaneously reachable states that share a control state, we add a conjunct to the μCLP query requiring that not all register values in the two simultaneously reachable states are equal. This eliminates quantifier instantiations that simplify to the trivially false formula: $\mathsf{prodreach}_{\mathtt{p}}(s, \boldsymbol{r}, s, \boldsymbol{r}) \;\wedge\; \varphi \wedge \mathsf{unreach}^{\varepsilon}_{\mathtt{p,q}}(s, \boldsymbol{r}, x)$.

Bugs found in MUVAL. While implementing SPROUT, we discovered a soundness bug in MUVAL's `parallel` and `parallel_exc` modes that led its output to depend on the order of least and greatest fixpoint predicates in μCLP instances containing only one kind of fixpoint. We also discovered a minor bug in MUVAL's constraint simplifier when optimizing queries containing negation or implication. Both bugs were reported to and subsequently fixed by MUVAL's developers.

4 Evaluation

All experiments in this section are run on a 2024 MacBook Air with an Apple M3 chip and 16GB of RAM. Verification times reported are the sum of GCLTS checking time and implementability checking time, with timeouts for individual μCLP instances specified separately.

4.1 Optimization Efficacy

We first evaluate the efficacy of SPROUT's optimizations, detailed in §3. We compare the verification times of SPROUT's pre-filtered, optimized μCLP instances against the naive encoding of definitions in [17]. We benchmark on examples of various sizes, measured by the number of transitions in the protocol specification. All examples are non-binary so as to reflect a difference in implementability checking time. The results in Table 1 show that naively encoding [17]'s conditions renders verification intractable for protocols with more than 2 transitions, and that SPROUT's optimizations yield a speedup by over two orders of magnitude.

4.2 Evaluation and Comparison Against SESSION*

Next, we evaluate SPROUT in terms of expressivity, precision and efficiency.

Example	$\lvert\mathcal{P}\rvert$	$\lvert\Delta\rvert$	Sprout	time	Naive [17]	time
figure12-yes	3	2	impl.	2.0s	impl.	2.4s
figure12-no	3	2	non-impl.	3.0s	non-impl.	2.3s
TwoBuyer	3	9	impl.	3.8s	timeout (300s)	311.2s
higher-lower-ultimate	3	9	impl.	11.1s	out of memory	610.4s
higher-lower-no	3	9	non-impl.	16.1s	non-impl.	349.8s
symbolic-two-bidder-yes	3	10	impl.	27.4s	timeout (300s)	648.4s
symbolic-two-bidder-no1	3	11	non-impl.	30.0s	out of memory	891.5s

Table 1: Comparison of verification times with and without optimizations.

Expressivity. To evaluate expressivity, we took the union of two benchmark suites from tools most closely related to Sprout: Session* [26] and Rumpsteak with refinements [25]. Both works target multiparty protocols with refinements, and in addition to checking implementability, generate type signatures against which user-provided local implementations can be statically type-checked. Session*'s benchmark suite contains 11 examples, all of which utilize refinements. Despite the title of [25], Rumpsteak's suite of 10 examples contains only 5 with refinements, and 4 that are multiparty, for a total of 2/10 multiparty examples with refinements. We omitted finite, binary protocols that can be handled by existing sound and complete tools for finite multiparty session types, such as [16], leaving us with 6 examples from Rumpsteak. Sprout was able to express all 17 examples from the literature. We then attempted to translate Session*'s examples into Rumpsteak's syntax, and vice versa, in an attempt to compare all three tools. Although both Session* and Rumpsteak adopt a Scribble-like syntax, we found that Session* could express all 6 of Rumpsteak's examples, whereas Rumpsteak could only express 3/11 of Session*'s examples, even after accommodating minor discrepancies that were immaterial to the high-level protocol intent. The key expressivity gap lay in the fact that Sprout and Session* both support loop recursion variables, e.g. in the two-bidder protocol, z_1 and z_2 that track $\mathtt{B}_1$ and $\mathtt{B}_2$'s respective last bids, whereas Rumpsteak does not.

Precision. The benchmark suites of both Session* and Rumpsteak exclusively contain implementable examples. In evaluating precision, we are interested in both the *soundness* and *completeness* of the tool: does it correctly accept implementable protocols, and correctly reject non-implementable ones? Thus, we expand our benchmark suite with a new set of examples based on protocols from prior works [5, 16, 17], where for each protocol we include *both* an implementable and non-implementable version. We also introduce implementable and non-implementable variations of common protocols in the literature (e.g. two-bidder, higher lower guessing game). Some of the non-implementable examples were inspired by bugs inadvertently introduced in the process of translating examples into Sprout, and most non-implementable examples have a small edit distance to their implementable counterpart. A short description of each new example and any bugs contained can be found at `https://github.com/nyu-acsys/sprout/tree/main/examples`. In translating our new examples to Session* and Rumpsteak, we found a similar pattern as before: Session* could express 20/21 examples, whereas Rumpsteak could only express 10/21.

`Calculator` was not expressible in Session* due to lack of support for multiplication, whereas `higher-lower-no`'s implementability bug was ruled out by Session*'s type checker.

The result of evaluating Session* and Sprout on the overall set of 37 examples is given in Table 2. We omitted evaluation results from Rumpsteak due to the tool's lack of formal guarantees and limited expressivity. To achieve a faithful comparison, verification times reported for Session* are only for checking projectability of global types and computing local types for each role.

Source	Example	$\|\mathcal{P}\|$	Impl.	Sprout	Time	Session*	Time
[26]	Calculator	2	✓	✓	0.6s	N/A	2.0s
	Fibonacci	2	✓	✓	0.5s	✓	1.8s
	HigherLower	3	✓	✓	15.2s	✓	3.9s
	HTTP	2	✓	✓	0.4s	✓	1.9s
	Negotiation	2	✓	✓	1.0s	✓	1.9s
	OnlineWallet	3	✓	✓	9.4s	✓	3.3s
	SH	3	✓	✓	237.1s	✓	5.6s
	Ticket	2	✓	✓	0.6s	✓	1.9s
	TravelAgency	2	✓	✓	9.2s	✓	3.1s
	TwoBuyer	3	✓	✓	3.8s	✓	2.8s
[25]	DoubleBuffering	3	✓	✓	1.5s	✓	2.3s
	OAuth	3	✓	✓	6.2s	✓	2.3s
	PlusMinus	3	✓	✓	5.2s	×	2.1s
	RingMax	7	✓	✓	3.7s	✓	4.7s
	SimpleAuth	2	✓	✓	0.5s	✓	2.0s
	TravelAgency2	2	✓	✓	1.7s	✓	1.8s
[16]	send-validity-yes	4	✓	✓	1.9s	×	2.1s
	send-validity-no	4	×	×	1.9s	×	2.1s
	receive-validity-yes	3	✓	✓	5.1s	×	2.3s
	receive-validity-no	3	×	×	3.6s	×	2.0s
[17]	symbolic-two-bidder-yes	3	✓	✓	27.4s	×	2.0s
	symbolic-two-bidder-no1	3	×	×	30.0s	×	2.1s
	figure12-yes	3	✓	✓	2.0s	✓	2.0s
	figure12-no	3	×	×	3.0s	✓	3.0s
	symbolic-send-validity-yes	4	✓	✓	6.5s	×	2.5s
	symbolic-send-validity-no	4	×	×	5.3s	×	2.6s
	symbolic-receive-validity-yes	3	✓	✓	6.6s	×	2.8s
	symbolic-receive-validity-no	3	×	×	7.6s	×	2.8s
[5]	fwd-auth-yes	3	✓	✓	10.3s	×	2.3s
	fwd-auth-no	3	×	?	T/O	×	2.2s
new	symbolic-two-bidder-no2	3	×	×	23.9s	×	2.8s
	higher-lower-ultimate	3	✓	✓	11.1s	×	2.4s
	higher-lower-winning	3	✓	?	T/O	✓	229.8s
	higher-lower-no	3	×	×	16.1s	N/A	2.2s
	higher-lower-encrypt-yes	4	✓	✓	9.3s	×	2.3s
	higher-lower-encrypt-no	4	×	×	177.3s	×	2.4s
	higher-lower-mixed	3	×	×	19.3s	×	2.3s

Table 2: Comparison of verification times with [26]. For each example, we report the number of participants ($|\mathcal{P}|$), ground truth implementability (✓ or ×), verification times for Session* [26] and Sprout with a 30s timeout per μCLP instance (T/O), and the result: ✓ for implementable/projectable, × for non-implementable/non-projectable, and ? for inconclusive due to timeout. Examples not expressible in Session* are marked with N/A.

The incompleteness of Session* is made apparent by our evaluation: of the 20 new examples expressible in Session*, containing an even mix of implementable and non-implementable protocols, Session* rejected all but 3/20. The source of incompleteness is twofold. For one, Session*'s notion of implementability is relative to *local types*, whose syntax a priori rules out communication patterns such as receiver choice from different senders. In contrast, [17] and Sprout's notion of implementability is relative to a more expressive semantic model, called communicating labeled transition systems [17, Definition 3.3]. For two, Session* implements the merge-based projection operator from [14]. This projection operator is inherently incomplete even for global types without refinements (see [16] for a detailed discussion), and thus the refinement type system presented in [26] inherits all sources of incompleteness. [‖]

Efficiency. In terms of efficiency, Session*'s verification times were mostly below 5s[**], whereas Sprout's verification times varied widely depending on the number of transitions in the protocol, and whether the protocol is binary. For binary protocols, the verification times of Sprout are competitive with those of Session*. For multiparty protocols, most examples returned in less than 10s, with the exception of 3 timeouts, whose timeout limits were set to 30s per μCLP instance. [††] As mentioned in §3, the verification bottleneck of Sprout lies in the efficiency of MuVal– instance generation introduces negligible overhead. The modularity of [17]'s Coherence Conditions means Sprout's efficiency could be improved by running all generated μCLP instances in parallel.

Acknowledgements. The authors thank Hiroshi Unno for his correspondences regarding MuVal. This work is supported in parts by the National Science Foundation under the grant agreement 2304758 and by the Luxembourg National Research Fund (FNR) under the grant agreement C22/IS/17238244/AVVA.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Software Eng. **29**(7), 623–633 (2003). https://doi.org/10.1109/TSE.2003.1214326, https://doi.org/10.1109/TSE.2003.1214326
2. Alur, R., Yannakakis, M.: Model checking of message sequence charts. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1664, pp. 114–129. Springer (1999). https://doi.org/10.1007/3-540-48320-9_10, https://doi.org/10.1007/3-540-48320-9_10

[‖]Note that Session*'s false positive result for `figure12-no` does not indicate the tool is unsound; rather, the user will fail to produce an implementation that typechecks against the generated local types because no implementation exists.

[**]Session* was run as a Docker container, and thus its verification times include emulation overhead.

[††]Note that when Sprout returns non-implementable for protocols containing instances that timeout, the verification time may increase directly with the timeout limit.

3. Bocchi, L., Demangeon, R., Yoshida, N.: A multiparty multi-session logic. In: Palamidessi, C., Ryan, M.D. (eds.) Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8191, pp. 97–111. Springer (2012). https://doi.org/10.1007/978-3-642-41157-1_7, https://doi.org/10.1007/978-3-642-41157-1_7
4. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6269, pp. 162–176. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_12, https://doi.org/10.1007/978-3-642-15375-4_12
5. Cruz-Filipe, L., Graversen, E., Lugovic, L., Montesi, F., Peressotti, M.: Functional choreographic programming. In: Seidl, H., Liu, Z., Pasareanu, C.S. (eds.) Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium, Tbilisi, Georgia, September 27-29, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13572, pp. 212–237. Springer (2022). https://doi.org/10.1007/978-3-031-17715-6_15, https://doi.org/10.1007/978-3-031-17715-6_15
6. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020). https://doi.org/10.1016/j.tcs.2019.07.005, https://doi.org/10.1016/j.tcs.2019.07.005
7. Gazagnaire, T., Genest, B., Hélouët, L., Thiagarajan, P.S., Yang, S.: Causal message sequence charts. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4703, pp. 166–180. Springer (2007). https://doi.org/10.1007/978-3-540-74407-8_12, https://doi.org/10.1007/978-3-540-74407-8_12
8. Genest, B., Muscholl, A.: Message sequence charts: A survey. In: Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France. pp. 2–4. IEEE Computer Society (2005). https://doi.org/10.1109/ACSD.2005.25, https://doi.org/10.1109/ACSD.2005.25
9. Genest, B., Muscholl, A., Peled, D.A.: Message sequence charts. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]. Lecture Notes in Computer Science, vol. 3098, pp. 537–558. Springer (2003). https://doi.org/10.1007/978-3-540-27755-2_15, https://doi.org/10.1007/978-3-540-27755-2_15
10. Genest, B., Muscholl, A., Seidl, H., Zeitoun, M.: Infinite-state high-level MSCs: Model-checking and realizability. J. Comput. Syst. Sci. **72**(4), 617–647 (2006). https://doi.org/10.1016/j.jcss.2005.09.007, https://doi.org/10.1016/j.jcss.2005.09.007
11. Gheri, L., Lanese, I., Sayers, N., Tuosto, E., Yoshida, N.: Design-by-contract for flexible multiparty session protocols. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 8:1–8:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.ECOOP.2022.8, https://doi.org/10.4230/LIPIcs.ECOOP.2022.8

12. Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: Multiparty languages: The choreographic and multitier cases (pearl). In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, vol. 194, pp. 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.22, https://doi.org/10.4230/LIPIcs.ECOOP.2021.22
13. Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022). https://doi.org/10.1145/3498684, https://doi.org/10.1145/3498684
14. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472, https://doi.org/10.1145/1328438.1328472
15. Li, E.: Sprout: A verifier for symbolic multiparty protocols (CAV'25 AE) (Apr 2025). https://doi.org/10.5281/zenodo.15313597, https://doi.org/10.5281/zenodo.15313597
16. Li, E., Stutz, F., Wies, T., Zufferey, D.: Complete multiparty session type projection with automata. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13966, pp. 350–373. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_17, https://doi.org/10.1007/978-3-031-37709-9_17
17. Li, E., Stutz, F., Wies, T., Zufferey, D.: Characterizing implementability of global protocols with infinite states and data. Proc. ACM Program. Lang. **9**(OOPSLA1), 1434–1463 (2025). https://doi.org/10.1145/3720493, https://doi.org/10.1145/3720493
18. Lohrey, M.: Realizability of high-level message sequence charts: closing the gaps. Theor. Comput. Sci. **309**(1-3), 529–554 (2003). https://doi.org/10.1016/J.TCS.2003.08.002, https://doi.org/10.1016/j.tcs.2003.08.002
19. Mauw, S., Reniers, M.A.: High-level message sequence charts. In: Cavalli, A.R., Sarma, A. (eds.) SDL '97 Time for Testing, SDL, MSC and Trends - 8th International SDL Forum, Evry, France, 23-29 September 1997, Proceedings. pp. 291–306. Elsevier (1997)
20. Morin, R.: Recognizable sets of message sequence charts. In: Alt, H., Ferreira, A. (eds.) STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2285, pp. 523–534. Springer (2002). https://doi.org/10.1007/3-540-45841-7_43, https://doi.org/10.1007/3-540-45841-7_43
21. Muscholl, A., Peled, D.A.: Message sequence graphs and decision problems on Mazurkiewicz traces. In: Kutylowski, M., Pacholski, L., Wierzbicki, T. (eds.) Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1672, pp. 81–91. Springer (1999). https://doi.org/10.1007/3-540-48340-3_8, https://doi.org/10.1007/3-540-48340-3_8
22. Roychoudhury, A., Goel, A., Sengupta, B.: Symbolic message sequence charts. ACM Trans. Softw. Eng. Methodol. **21**(2), 12:1–12:44 (2012). https://doi.org/10.1145/2089116.2089122, https://doi.org/10.1145/2089116.2089122

23. Toninho, B., Yoshida, N.: Certifying data in multiparty session types. J. Log. Algebraic Methods Program. **90**, 61–83 (2017). https://doi.org/10.1016/J.JLAMP.2016.11.005, https://doi.org/10.1016/j.jlamp.2016.11.005
24. Unno, H., Terauchi, T., Gu, Y., Koskinen, E.: Modular primal-dual fixpoint logic solving for temporal verification. Proc. ACM Program. Lang. **7**(POPL), 2111–2140 (2023). https://doi.org/10.1145/3571265, https://doi.org/10.1145/3571265
25. Vassor, M., Yoshida, N.: Refinements for multiparty message-passing protocols: Specification-agnostic theory and implementation. In: Aldrich, J., Salvaneschi, G. (eds.) 38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria. LIPIcs, vol. 313, pp. 41:1–41:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/LIPICS.ECOOP.2024.41, https://doi.org/10.4230/LIPIcs.ECOOP.2024.41
26. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA), 148:1–148:30 (2020). https://doi.org/10.1145/3428216, https://doi.org/10.1145/3428216