




Taming Concurrency for Verification Using Multiparty Session Types

Kirstin Peters^{1,2}(✉) , Christoph Wagner¹, and Uwe Nestmann¹ 

¹ TU Berlin, Berlin, Germany

² TU Darmstadt, Darmstadt, Germany

kirstin.peters@cs.tu-darmstadt.de

Abstract. The additional complexity caused by concurrently communicating processes in distributed systems render the verification of such systems into a very hard problem. Multiparty session types were developed to govern communication and concurrency in distributed systems. As such, they provide an efficient verification method w.r.t. properties about communication and concurrency, like communication safety or progress. However, they do not support the analysis of properties that require the consideration of concrete runs or concrete values of variables. We sequentialise well-typed systems of processes guided by the structure of their global type to obtain interaction-free abstractions thereof. Without interaction, concurrency in the system is reduced to sequential and completely independent parallel compositions. In such abstractions, the verification of properties such as e.g. data-based termination that are not covered by multiparty session types, but rely on concrete runs or values of variables, becomes significantly more efficient.

Keywords: Concurrency · Verification · Multiparty session types

1 Introduction

Modern society is increasingly dependent on large-scale software systems that are distributed, collaborative, and communication-centred. One of the techniques developed to handle the additional complexity caused by distributed actors are *multiparty session types* (MPST) [19]. MPST allow to specify the desired behaviour of communication protocols as by-design correct types that are used to verify the communication structure of software products. The properties guaranteed by well-typed processes cover communication safety (all processes conform to globally agreed communication protocols) and liveness properties such as deadlock-freedom. Their main advantage is that their verification method is extremely efficient—in comparison to e.g. standard model checking.

MPST were developed to govern communication and concurrency in distributed systems. However, as it is typical for type systems, standard MPST variants (without dependable types) do not support the analysis of properties that require the consideration of concrete runs or concrete values of variables.

The hardest part about the verification of distributed systems is the state space explosion that results from concurrent communication attempts, i. e., the exponential blow-up that results from computing all possible combinations of potential communication partners. The problem of concurrency mainly lies in the communication structure, which is already completely captured by MPST. We show that the knowledge of a program/system to be well-typed, allows us to sequentialise it following the structure of its global type and thereby to remove all communication. Accordingly, we show how we can benefit from the effort we spend on an MPST analysis of a system also for the verification of its properties that go beyond its communication structure.

We use the global type of a well-typed system to guide its sequentialisation. We refer to the result as *sequential global process* (SGP), although it might still contain parallel compositions, albeit only on completely independent parts. Since the structure of communication was already verified by the well-typedness proof, we can reduce communication to value updates. More precisely, we map well-typed systems that interact concurrently, to SGP-systems without any interaction mechanisms or name binders. Such SGP-systems consist of a vector of variables with values and a SGP-process that simulates the data flow of the original system. Therefore, we translate the reception of data in communication into updates of the vector in the SGP-system. By removing the communication we remove also the problem of state space explosion. Our translation is valid if the considered process is well-typed w. r. t. a (set of) global type(s). Thereby, we sequentialise communications that may happen concurrently in the original system but are sequential in global types. Note that such communications are always causally independent of each other, thus ordering them does not significantly influence the behaviour of the system, e. g. it does not influence what values are computed. Apart from such sequentialisations the original system and its abstraction into a SGP-system behave similarly.

Contributions. We provide an algorithm to remove communication from well-typed systems and thereby sequentialise them, while preserving the evolution of data of the original system. Deriving this algorithm was technically challenging but the result is a simple rewriting function and easy to automate.

Then we prove that, provided that the original system was well-typed, the algorithm produces a SGP-system that is closely related to the original system: the original system and its abstraction are related by a variant of operational correspondence [14] and are coupled similar [23]. With that, the derived SGP-system is a good abstraction of the original system that can be used instead of the original to verify properties on concrete data. Since the mapping into SGP-systems is usually linear and because SGP-systems do not contain any form of interaction or binders, properties can be checked more efficiently.

Finally, we provide a mapping—that is again a simple rewriting algorithm—from SGP-processes into *Promela*, the input language of the model checker *Spin* [16, 17]. With that, the properties that are not already guaranteed by the MPST analysis but require the consideration of concrete runs or concrete data can be checked. Since the main challenge here is the sequentialisation of concurrent

systems into interaction-free abstractions, the translation of SGP-systems into *Promela* is simple and can be used as a role model to obtain similar mappings for other model checkers.

Related Work. Intuitively, the technique that we present in this paper is a special case of partial order reduction (compare e. g. to [24]) as they can be found in model checkers. This technique tries to reduce the state space that has to be inspected in verification, by identifying different sequences of transitions that lead to similar states. Here, instead of searching for such similar states, we follow the structure of the global type, where well-typedness ensures that the generated abstraction captures the complete state space of the original system modulo coupled similarity.

The approach of [1] is very similar to this paper. Just as our algorithm, they rewrite a program (written in *Haskell*) by replacing communication with value updates, to obtain a sequential abstraction of the program on that verification—e. g. of termination based on values that are computed at runtime—can be done efficiently. The main difference is that [1] requires that the considered programs satisfy *symmetric non-determinism* whereas we require that programs are well-typed using MPST. Assuming asynchronous communication, symmetric non-determinism means that every receive in a given program location can receive only messages from either a single process or a set of symmetric processes, i. e., processes running the same code, at the same program location. MPST are more flexible, i. e., are not limited to systems that satisfy symmetric non-determinism. Hence, the method presented here can be applied to a larger class of programs.

Interestingly, we find a similar idea also in papers about the verification of distributed algorithms via invariants that use so-called *standard forms* (compare e. g. to [9,29]), where the global view gets constructed by gathering and combining all local processes. In case of [29] standard forms have their own TLA-like semantics that is 1-to-1 correspondent to the calculus semantics for proving properties on data. The main difference to these approaches is that we completely remove communication and present an algorithm to automatically derive this global view from a given well-typed system and its global type.

In [6] global types are translated into processes to mediate between multi-party and binary session types. These mediator processes capture the behaviour of global types—w. r. t. the communication structure and not values—to provide a disciplined communication exchange that allows to translate MPST into binary sessions. In contrast to this approach, we map processes onto processes and use global types to guide this mapping, where the communication structure is removed and our focus is on the evolution of data.

Choreographies [22] are global descriptions of distributed systems from which the distributed system is generated by endpoint projection. In contrast, we start with the distributed system and its global type. Note that global types describe solely the communication structure, i. e., interactions, of the system and do not contain any other implementation details of single peers. With that, MPST have an advantage in comparison to choreographies in industrial settings,

where different parts are developed independently. Moreover, we also consider the case of interleaved MPST sessions. Our challenge is to derive a global description on how the data evolves in the system. This is related to the extraction of choreographies from distributed systems as discussed in [7, 10]. However, without the global type as guide, the described algorithms to extract choreographies are exponential, whereas our algorithm is usually linear.

In [5] MPST are extended by assertions that allow to verify properties on data values provided that these properties are satisfied in all runs. In contrast, our approach allows to efficiently compute the exact values that are computed in concrete runs. Moreover, the language of assertions is limited to the language defined in [5] and an extension might require to redo some proofs, whereas here only the translation into *Promela*, i. e., the use of a concrete model checker, forces us to limit the languages of expressions and properties. The algorithm to sequentialise systems into SGP-systems does not rely on such limitations. A prominent example of a property that cannot be analysed statically is termination of a loop after computing some value. To prove such properties, a type system can use dependent types such as e. g. in [28]. In contrast to such extensions of MPST with dependable types, we do not add any complexity to the type system (or provide any new variant of MPST). Instead we provide a simple rewriting algorithm that transforms a well-typed system (after the type check) into an abstraction on that remaining properties on data can be verified with existing specialised tools.

Overview. In Sect. 2 we introduce multiparty session types very briefly. Section 3 describes how well-typed systems are sequentialised. Section 3.1 introduces a calculus for sequential global processes, Sect. 3.2 describes an algorithm to map into SGP-systems for the case of synchronous MPST and single sessions, and Sect. 3.3 discusses asynchronous variants of MPST and extends the algorithm to cover interleaved sessions. Section 4 shows operational correspondence and relates the original system and its abstraction by coupled similarity. Then, Sect. 5 illustrates how the sequentialisation can be used to verify properties of the original system. It discusses the limits of this method, i. e., what kind of properties cannot be analysed this way and presents a mapping from SGP-processes into *Promela*. We conclude in Sect. 6. Missing proofs and additional material can be found in [26].

2 Multiparty Session Types in a Nutshell

Our aim is to use the structure of a global type to remove communication—and with that the related concurrency—from the problem of verifying properties on the evolution of values. We conjecture that this procedure can be used for all kind of MPST variants but explain the method on a simple variant of synchronous MPST w. r. t. a single session. Later we extend our algorithm to asynchronous MPST with interleaved sessions. To explain the basic idea, we use a variant of multiparty synchronous session types as introduced in [2] with some alternations similar to variants as e. g. in [4, 11, 30].

MPST were developed to govern communication and concurrency in distributed systems. Therefore, systems are checked against a *global type*. Global types specify the desired communication structure from a global point of view. The specified communication structure of a global type describes a *session* and the participants of such a session are called *roles*. Here, they are given by

$$G ::= r_1 \rightarrow r_2 : \left\{ l_i \langle \tilde{U}_i \rangle . G_i \right\}_{i \in I} \quad | \quad G_1, G_2 \quad | \quad (\mu t) G \quad | \quad t \quad | \quad \text{end}$$

The first construct specifies a communication from Role r_1 to r_2 that offers different branches for the receiver with respect to a label l_i that is transmitted by the sender, where \tilde{U}_i are the sorts (i. e., base types) of the transmitted values. If I is a singleton, we abbreviate communication with $r_1 \rightarrow r_2 : l \langle \tilde{U} \rangle . G$. The other constructs introduce parallel composition, recursion, and successful termination.

The systems, that we want to analyse, are modelled in a *session calculus*. As usual, we use an extension of the π -calculus [21] given by

$$P ::= \bar{a}[2..n](s).P \quad | \quad a(s[r]).P \quad | \quad s[r_1, r_2]!l(\tilde{e}).P \quad | \quad s[r_2, r_1]? \{l_i(\tilde{x}_i).P_i\}_{i \in I} \\ | \quad \text{if } c \text{ then } P_1 \text{ else } P_2 \quad | \quad P_1 | P_2 \quad | \quad \mathbf{0} \quad | \quad (\nu s)P \quad | \quad (\mu X)P \quad | \quad X$$

The first two constructs are used to initiate a session. The next two constructs model the sender and the receiver of a communication within a session, where the \tilde{x}_i are (*input bounded*) *variables* that are instantiated as result of a communication by the received values. The remaining constructs introduce conditionals, parallel composition, termination, restriction, and recursion. Since we want to use the model checker **Spin** later, we restrict expressions e (the values that are transmitted in communication) and conditions c (used to guide conditionals) to functions that are known by **Promela**, the input language of **Spin**.

We use structural congruence (\equiv) to abstract from syntactically different but semantically similar processes, where \equiv is the least congruence that satisfies alpha-conversion (\equiv_α) and the rules:

$$P | \mathbf{0} \equiv P \quad P_1 | P_2 \equiv P_2 | P_1 \quad P_1 | (P_2 | P_3) \equiv (P_1 | P_2) | P_3 \\ (\mu X)P \equiv P \{(\mu X)P/X\} \quad (\nu s)(\nu s')P \equiv (\nu s')(\nu s)P \quad (\nu s)\mathbf{0} \equiv \mathbf{0} \\ (\nu s)(P_1 | P_2) \equiv P_1 | (\nu s)P_2 \quad \text{if } s \notin \text{fn}(P_1)$$

The reduction semantics of the session calculus is given by the rules:

$$\text{(Link)} \frac{}{\bar{a}[2..n](s).P_1 | a(s[2]).P_2 | \dots | a(s[n]).P_n \mapsto (\nu s)(P_1 | P_2 | \dots | P_n)} \\ \text{(Com)} \frac{j \in I}{s[r_1, r_2]!l_j(\tilde{e}).P | s[r_2, r_1]? \{l_i(\tilde{x}_i).P_i\}_{i \in I} \mapsto P | (P_j \{ \tilde{e}/\tilde{x}_j \})} \\ \text{(If-T)} \frac{c}{\text{if } c \text{ then } P_1 \text{ else } P_2 \mapsto P_1} \quad \text{(If-F)} \frac{\neg c}{\text{if } c \text{ then } P_1 \text{ else } P_2 \mapsto P_2} \\ \text{(Par)} \frac{P_1 \mapsto P'_1}{P_1 | P_2 \mapsto P'_1 | P_2} \quad \text{(Res)} \frac{P \mapsto P'}{(\nu s)P \mapsto (\nu s)P'} \\ \text{(Struc)} \frac{P_1 \equiv P_2 \quad P_2 \mapsto P'_2 \quad P'_2 \equiv P'_1}{P_1 \mapsto P'_1}$$

The Rule **Link** initialises a session s on the roles $1, \dots, n$, where 1 requested the session on channel a and each i participates in the session as P_i . Communication within a session s is described by Rule **Com**, where in the case of matching roles and labels the continuations of sender and receiver are unguarded and the variables \tilde{x} are replaced by the values \tilde{e} in the receiver. The Rules **If-T** and **If-F** reduce conditionals as expected. The remaining rules allow for steps in various contexts and are standard.

Let $r(\cdot)$ return the roles used in a global type or a process. A process P has an *actor* on $c[r_1]$ if P has an unguarded subterm of the form $\bar{c}[2..n](s).P$ with $r_1 = 1$ or $c(s[r_1]).P$ (for session invitations) or an unguarded subterm of the form $c[r_1, r_2]!(\tilde{e}).P$ or $c[r_1, r_2]? \{!_i(\tilde{x}_i).P_i\}_{i \in I}$ (for communication). Let $\text{act}(P)$ be set of actors in P . If unambiguous, i. e., if there is only one session, we omit the session channel and abbreviate actors by their role.

The processes P are checked against their specification in *type judgements* $\Gamma \vdash P \triangleright \Delta$, where Γ, Δ are *type environments* that are built from the type information in the global type. A system that passes such a type check is denoted as *well-typed*. The design of MPST guarantees strong properties for the communication structure of well-typed systems.

Theorem 1. *Assume $\Gamma \vdash P \triangleright \Delta$, i. e., P is well-typed.*

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

Linearity: *P has no two unguarded senders/receivers for the same actor.*

Progress: *If $P \mapsto^* P'$ then either $P' \equiv \mathbf{0}$ or $P' \mapsto P''$.*

To prove these properties, we have to reason about the *typing rules* that define under which circumstances a type judgement is valid. Due to space limitations, the typing rules as well as some other important aspects of MPST (e. g. projection and local types) and the proofs are postponed to [26]. Note that we do *not* introduce a new variant of MPST. Instead we rely on a standard MPST variant of that we introduced global types and the session calculus, because they are necessary to understand the remainder of this paper.

3 Sequentialising Well-Typed Systems

MPST are designed to analyse the communication structure of a system. Well-typed systems are guaranteed to satisfy properties like communication safety or progress. What remains, are safety and liveness properties that involve data.

We use the global type of a well-typed term to guide the sequentialisation of the implementation. The result is a kind of process that we call *sequential global process* (SGP), although it might still contain parallel compositions but only on completely independent parts. This abstraction of the implementation allows us to analyse properties on the values of data in the implementation without the problem of state space explosion that is caused by the concurrency of communication in the original system.

$$\begin{array}{c}
\text{(Ass)} \frac{}{\langle \mathcal{V}; \tilde{v} := \tilde{e}.S \rangle \mapsto \text{eval}(\langle \mathcal{V}(\tilde{v}) := \tilde{e}; S \rangle)} \quad \text{(Par)} \frac{\langle \mathcal{V}; S_1 \rangle \mapsto \langle \mathcal{V}'; S'_1 \rangle}{\langle \mathcal{V}; S_1 \parallel S_2 \rangle \mapsto \langle \mathcal{V}'; S'_1 \parallel S_2 \rangle} \\
\text{(If-T)} \frac{c}{\langle \mathcal{V}; \text{if } c \text{ then } S_1 \text{ else } S_2 \rangle \mapsto \langle \mathcal{V}; S_1 \rangle} \quad \text{(If-F)} \frac{\neg c}{\langle \mathcal{V}; \text{if } c \text{ then } S_1 \text{ else } S_2 \rangle \mapsto \langle \mathcal{V}; S_2 \rangle} \\
\text{(Struc)} \frac{S_1 \equiv_S S_2 \quad S_2 \mapsto S'_2 \quad S'_2 \equiv_S S'_1}{S_1 \mapsto S'_1}
\end{array}$$

Fig. 1. Reduction Semantics of SGP-Systems.

3.1 A Calculus for Sequential Global Processes

SGP-processes are simple processes consisting of assignments of values to variables, conditionals, parallelism, termination, and recursion.

Definition 1 (SGP-Processes). *SGP-processes are given by*

$$S ::= \tilde{v} := \tilde{e}.S \quad | \quad \text{if } c \text{ then } S_1 \text{ else } S_2 \quad | \quad S_1 \parallel S_2 \quad | \quad \mathbf{0} \quad | \quad (\mu X)S \quad | \quad X$$

where \tilde{e} are expressions to calculate a value, c are boolean conditions, and X process variables.

SGP-processes introduce a new operator to assign values to variables in a vector. An assignment $\tilde{v} := \tilde{e}.S$ describes a SGP-process that updates the variables \tilde{v} by the values \tilde{e} and then continues as S , where $\tilde{v} := \tilde{e}.S$ is short hand for $(v_1, \dots, v_n) := (e_1, \dots, e_n).S$. If \tilde{v} (and accordingly also \tilde{e}) is the empty sequence, then we abbreviate this empty assignment by $\tau.S$. Note that SGP-processes inherit the parallel operator not from processes but from global types. Thus, the parallel composition $S_1 \parallel S_2$ describes that S_1 and S_2 are independent, i. e., all variables that appear on both sides are used as read-only on both sides. The remaining operators for conditionals, successful termination, and recursion are inherited from processes. Note that SGP-processes do neither contain any interaction mechanisms nor name binders. But we still have branching via conditionals and recursion.

The SGP-processes are combined with a vector \mathcal{V} of variables, that represents the current values of the local variables of all processes of the original distributed system. They consist of the input bounded variables of the implementation. A *SGP-system* $\langle \mathcal{V}; S \rangle$ then consists of a knowledge vector \mathcal{V} and a SGP-process S .

Structural congruence on SGP-processes \equiv_S is the restriction of \equiv on SGP-processes. Let \equiv_S be the least congruence that satisfies the rules $S \equiv_S \text{eval}(S)$ and $\langle \mathcal{V}; S \rangle \equiv_S \langle \mathcal{V}; S' \rangle$ if $S \equiv_S S'$. We write $\mathcal{V}(\tilde{v}) := \tilde{e}$ for the result of replacing, for all $v_i \in \tilde{v}$, the current value of the variable v_i in the vector \mathcal{V} by the value that results from the evaluation of the expressions e_i . The semantics of SGP-systems is given in Fig. 1. We naturally extend substitution to SGP-systems, i. e., $\langle \mathcal{V}; S \rangle \sigma = \langle \mathcal{V}\sigma; S\sigma \rangle$. Let $\text{eval}(\langle \mathcal{V}; S \rangle)$ be the result of replacing all variables

v in conditions and expressions in S that are not sequentially hidden after an assignment of v by the current value of v in \mathcal{V} , e. g. :

$$\begin{aligned}
& \text{eval}(\langle (v = 5); \text{if } v > 6 \text{ then } \mathbf{0} \text{ else } v := v + 1.v := v + 1.\mathbf{0} \rangle) \\
&= \langle (v = 5); \text{if } 5 > 6 \text{ then } \mathbf{0} \text{ else } v := 5 + 1.v := v + 1.\mathbf{0} \rangle \\
&\mapsto \langle (v = 5); v := 5 + 1.v := v + 1.\mathbf{0} \rangle \\
&\mapsto \text{eval}(\langle (v = 6); v := v + 1.\mathbf{0} \rangle) = \langle (v = 6); v := 6 + 1.\mathbf{0} \rangle \\
&\mapsto \text{eval}(\langle (v = 7); \mathbf{0} \rangle) = \langle (v = 7); \mathbf{0} \rangle
\end{aligned}$$

3.2 Mapping Well-Typed Systems onto SGP-Systems

We use the global type of a well-typed process P to sequentialise P into a SGP-process. Because of the parallel operator, SGP-processes are not completely sequential. However, since we remove communication and with it all forms of interaction from SGP-processes, parallel composition in SGP-processes is between independent parts only. More precisely, SGP-systems cover read and write operations on their vector of variables that simulate the evolution of knowledge in the original distributed system.

The main idea of the algorithm is simple. We fuse matching senders and receivers, i. e., the receiver that receives a message with the sender that transmitted this message, into a single SGP value assignment. The value assignment captures what the processes gain as new information from a communication. The problem is that finding the matching communication partners in the general π -calculus is NP-hard. In the π -calculus it is possible to have several matching receivers for a single sender or vice versa. Performing a communication step can unguard further senders and receivers. So different choices of matching pairs of communication partners and different orders in that communications are performed influence the further behaviour. To reduce the complexity of this problem, we use the type information that allows us to completely avoid the search for matching communication partners.

Firstly, well-typedness guarantees that there are no races at runtime, i. e., in no state there is more than one matching receiver for a sender and vice versa. This ensures, that for each well-typed system there is indeed a single SGP-abstraction that captures its overall behaviour, whereas without well-typedness (i. e., in the presence of races) several SGP-abstractions might be necessary to describe the behaviour of a single system. Secondly, well-typedness also ensures that there are no orphan communication partners, i. e., each sender will eventually meet a matching receiver and vice versa. Finally, the global type of a well-typed system tells us when and where communication takes place. Or, more precisely, the global type tells us one possible order of the communications and well-typedness ensures that all other possible orderings of communications of the system are similar (see Sect. 4). Accordingly, we do not search for matching communication partners but follow the structure of the global type. If the global type specifies that next there is a communication then we know that in the mentioned actors the respective send and receive action is indeed unguarded or guarded only by conditionals, that can be resolved without interactions.

Similarly, it is difficult in general π -calculussystems to identify at which point we have to introduce a global loop to translate the recursive behaviour of the single actors into a recursion of the global abstraction. Again we follow the structure of the global type and simply introduce a global loop if the global type loops, while ignoring the structure of recursion in the actors and only unfolding local recursion if necessary.

When we remove communication prefixes in order to obtain a SGP-process, we lose their respective scopes. To avoid ambiguities in SGP-systems and to clarify the owner of variables, we indicate input bounded variables, i.e., the variables a SGP-process may write on, by its corresponding actor. The variables indicated by an actor are the local knowledge of this actor. SGP-processes that are derived from well-typed processes will not have write access on variables of other actors but may read them to perform value updates.

The following mapping relies on the fact that the parallel composition $\prod_{i \in I} P_i$ is well-typed w. r. t. the global type G . We prove in Theorem 2 below, that this mapping indeed produces a SGP-process in this case.

Definition 2. *The partial mapping $\text{SGP}(\{P_i\}_{i \in I}, G)$ is defined inductively as:*

1. $\mathbf{0}$, if $G = \text{end}$
2. X_t , else if $G = t$
3. $\text{SGP}\left(\{P'_j\} \cup \{P_i\}_{i \in I \setminus \{j\}}, G\right)$,
else if there is some $j \in I$ such that $P_j = (\nu s)P'_j$
4. $\text{SGP}\left(\{P_{j1}, P_{j2}\} \cup \{P_i\}_{i \in I \setminus \{j\}}, G\right)$,
else if there is some $j \in I$ such that $P_j = P_{j1} \mid P_{j2}$
5. $\text{SGP}\left(\{P'_j\}_{(\mu X)P'_j/X}\} \cup \{P_i\}_{i \in I \setminus \{j\}}, G\right)$,
else if there is $j \in I$ such that $P_j = (\mu X)P'_j$
6. $\tilde{x}_m @s[r] := \tilde{e}. \text{SGP}\left(\{Q_m \{\tilde{x}_m @s[r]/\tilde{x}_m\}, Q\} \cup \{P_i\}_{i \in I \setminus \{k,l\}}, G_m\right)$,
else if there are $k, l \in I, m \in J \subseteq J'$ such that

$$G = r_1 \rightarrow r_2 : \left\{ l_j \langle \tilde{U}_j \rangle . G_j \right\}_{j \in J'}, P_k = s[r_1, r_2] !!_m(\tilde{e}) . Q,$$
and $P_l = s[r_2, r_1] ? \{ l_j(\tilde{x}_j) . Q_j \}_{j \in J'}$
7. $\text{SGP}(\{P_i\}_{i \in I_1}, G_1) \parallel \text{SGP}(\{P_j\}_{j \in I_2}, G_2)$,
else if there are some $I_1 \cup I_2 = I$ such that $G = G_1, G_2,$
 $\bigcup_{i \in I_1} r(P_i) = r(G_1)$, and $\bigcup_{j \in I_2} r(P_j) = r(G_2)$
8. $(\mu X_t) \text{SGP}(\{P_i\}_{i \in I}, G')$, else if $G = (\mu t)G'$
9. $\tau. \text{SGP}(\{P'_1, \dots, P'_n\}, G)$,
else if $\{P_i\}_{i \in I} = \{\bar{a}[2..n](s).P'_1, a[s[2]].P'_2, \dots, a[s[n]].P'_n\}$
10. if c then $\text{SGP}(\{P_{j1}\} \cup \{P_i\}_{i \in I \setminus \{j\}}, G)$ else $\text{SGP}(\{P_{j2}\} \cup \{P_i\}_{i \in I \setminus \{j\}}, G)$,
else if there is some $j \in I$ such that $P_j = \text{if } c \text{ then } P_{j1} \text{ else } P_{j2}$

Note that the different cases of this definition are ordered. Thus, a conditional is not resolved (Case 10) unless none of the other cases can be applied. The first two cases provide the base cases for global types that are terminated (Case 1) or

reduced to a type variable (Case 2). In these two cases the considered processes are ignored. The next three cases do not alter the type but prepare processes by removing restriction on session channels (Case 3)—which is safe because we will also remove all communication prefixes—splitting parallel compositions (Case 4), and unfolding recursion (Case 5). Because we require that process variables are guarded by a communication prefix, we cannot unfold the same recursion twice without applying another case in between.

The next three cases map processes that are well-typed w.r.t. a global type on communication (Case 6), parallel composition (Case 7), and recursion (Case 8), i. e., here we follow the structure of the global type to map the process. Case 6 unifies the sender and the receiver of a communication specified in the global type and maps it on corresponding value assignments. These assignments simulate the reception of the values \tilde{e} on the variables $\tilde{x}_m@s[r]$ of the receiver $s[r]$, where $\tilde{x}@s[r] = x_{1\check{s}[r]}, \dots, x_{n\check{s}[r]}$ is the result of indicating the variables with the actor $s[r]$ of the receiving end. The substitution of \tilde{x}_m into $\tilde{x}_m@s[r]$ ensures that names of different parallel branches are not confused.

This substitution does not remove all remaining name clashes but only the harmful clashes between parallel composed components of the considered system. Sequential composed input binders on the same variable and of the same actor are translated to the same name. If we apply this algorithm on processes that are not well-typed, we may still have parallel occurrences of syntactically the same name in parallel composed input binders. But well-typedness ensures that all such occurrences are linked to different actors and are thus distinguished. With that, we unify variables—that might have been denoted the same on purpose—and reduce the vector of variables in the SGP-system. Also, input bounded variables of different iterations of recursion are unified.

Case 7 maps a parallel composition of global types on a parallel composition of SGP-processes. Note that in both cases, parallel composition is between independent objects that have no means of interaction. To split the parallel components of the system accordingly, we rely on their roles. Well-typedness of the system ensures that it can be split as required.

Case 8 introduces recursion if the global type tells us to do so. This case does not alter the considered system or enforces any requirements on the structure of the system. Well-typedness ensures that the structure of the system w.r.t. recursion matches the recursion of the global type, but not necessarily that the system and the global type use recursion at the same time. For example $\bar{a}[2](s).(\mu X) s[1, 2]!!\langle 5 \rangle.s[1, 2]?l'(x).X \mid a(s[2]).(\mu X) s[2, 1]?l'(x).s[2, 1]!!\langle 6 \rangle.X$ is well-typed w.r.t. $1 \rightarrow 2 : l\langle \mathbb{N} \rangle.(\mu t) 2 \rightarrow 1 : l'\langle \mathbb{N} \rangle.1 \rightarrow 2 : l\langle \mathbb{N} \rangle.t$, although the global type partially unfolds the recursion in comparison to the recursion of the process. Therefore, we rely on well-typedness and use only the global type to determine the correct place of recursion, where Case 5 allows to unfold recursion in processes. To ensure that the process variables of nested recursions are not confused, the Cases 2 and 8 use the type variable of the global type as index to distinguish process variables.

Case 9 unifies session invitations and the corresponding acceptances and maps them on an empty value assignment τ . The session invitation mechanism

is already validated in the well-typedness proof and does not influence the data of the processes in the system. Thus, it is safe to ignore this step in SGP-processes.

Case 10 maps a conditional of one of the parallel components of the system to a conditional in the SGP-process. Global types do not consider conditionals of processes, but well-typedness ensures that both cases of the conditional have to follow the same global type. Because of that, both cases of the SGP-conditional inherit the same global type. To avoid unnecessary branching, we delay the mapping of conditionals until this is necessary e. g. to unguard a sender or receiver of a communication that is specified in the global type.

The mapping in Definition 2 is not deterministic, because it does not enforce an order in that several unguarded conditionals are mapped in Case 10 and this leads to different possible SGP-processes. Similarly, it is not specified in which order several restrictions in Case 3, several parallel compositions in Case 4, or several recursive processes in Case 5 are handled, though different orders in these cases will not lead to different SGP-processes. The other cases are guided by the global type or the fact that there is only one session. To obtain a deterministic version—and simplify the proofs—and to minimize the size of the computed SGP-process, we assume that Definition 2 gives precedence to parallel branches that implement (1) senders and (2) receivers that are unguarded in the global type and (3) smaller roles. However, different orders in that conditionals are handled lead to weakly bisimilar SGP-processes, because only unguarded conditionals are mapped, the translated subprocesses of the conditional are guarded by the resulting SGP-conditional, and an unguarded conditional will reduce to the same case in a single τ -step regardless of when we perform this step.

3.3 Asynchrony and Interleaved Sessions

The mapping in Definition 2 is designed for a synchronous variant of multiparty session types and only single sessions, because the syntax and semantics is simpler in these cases. However, the mapping in Definition 2 is *exactly the same* for the case of multiparty *asynchronous* session types as introduced in [19,20].

Note that the semantics of the session calculus defined in [19,20] use message queues to reflect the asynchronous nature of communication. Sending and receiving are decoupled into two separate steps to transmit and then read from message queues. Nonetheless, when we remove communication in the mapping $\text{SGP}(\cdot, \cdot)$, we unify sending and receiving into value assignments as described in the Case 6 of Definition 2. This is because, SGP-processes are designed to track the evolution of data values of processes and therefore only the reception of values is relevant. Intuitively, value assignments of SGP-processes reflect the case that a participant of a session has learned new information by the reception of values and this information flow is covered by value assignments. To determine the correct point in the behaviour of the system in that a particular participant gains new information through the reception of values, we rely on the fact that for this communication to happen both communication partners, the sender and the receiver have to be unguarded. Well-typedness and the structure of the global type, guide us in the case of concurrently enabled communication prefixes.

The definition of well-typed processes for several interleaved sessions is more difficult. As described in [3], we have to ensure that actions of different sessions do not cause deadlocks by cyclic dependencies. Processes with only acyclic dependencies between interactions of different sessions are denoted as *globally progressing*. However, adapting $\text{SGP}(\cdot, \cdot)$ to allow for several interleaved sessions in processes that are globally progressing is straightforward. First we remove name clashes between session channels using alpha conversion. Then we adapt the mapping $\text{SGP}(\cdot, \cdot)$ of Definition 2 into $\text{SGP}'(\cdot, \cdot)$, where the latter expects a set $\{P_i\}_{i \in I}$ and a set $\{(G_j, s_j)\}_{j \in J}$ of pairs of global types and session channels as input such that the parallel composition $\prod_{I \in I} P_i$ is well-typed w. r. t. $\{(G_j, s_j)\}_{j \in J}$ and $\prod_{I \in I} P_i$ does not contain name clashes between session channels.

Definition 3. $\text{SGP}'\left(\{P_i\}_{i \in I}, \{(G_j, s_j)\}_{j \in J}\right)$ is defined inductively as:

1. (a) $\mathbf{0}$, if $J = \emptyset$
 (b) $\text{SGP}'\left(\{P_i\}_{i \in I}, \{(G_j, s_j)\}_{j \in J \setminus \{k\}}\right)$,
else if there is some $k \in J$ such that $G_k = \text{end}$
2. X_G , else if $G = \{G_j\}_{j \in J} = \{t_j\}_{j \in J}$
3. $\text{SGP}'\left(\{P'_k\} \cup \{P_i\}_{i \in I \setminus \{k\}}, \{(G_j, s_j)\}_{j \in J}\right)$,
else if there is $k \in I$ such that $P_k = (\nu s)P'_k$
4. $\text{SGP}'\left(\{P_{k1}, P_{k2}\} \cup \{P_i\}_{i \in I \setminus \{k\}}, \{(G_j, s_j)\}_{j \in J}\right)$,
else if there is some $k \in I$ such that $P_k = P_{k1} \mid P_{k2}$
5. $\text{SGP}'\left(\{P'_k \{(\mu X)P'_k/X\}\} \cup \{P_i\}_{i \in I \setminus \{k\}}, \{(G_j, s_j)\}_{j \in J}\right)$,
else if there is $k \in I$ such that $P_k = (\mu X)P'_k$
6. $\tilde{x}_n @ s[r] := \tilde{e}. \text{SGP}'(\mathcal{P}, \mathcal{G})$ with $\mathcal{P} = \{Q_n \{\tilde{x}_n @ s[r] / \tilde{x}_n\}, Q\} \cup \{P_i\}_{i \in I \setminus \{m, o\}}$ and $\mathcal{G} = \{(G_{l,n})\} \cup \{(G_j, s_j)\}_{j \in J \setminus \{L\}}$,
else if there are $m, o \in I, l \in J, n \in K \subseteq K'$ such that

$$G_l = r_1 \rightarrow r_2 : \left\{ \! \left\langle \! \left\langle \tilde{U}_k \right\rangle \! \right\rangle . G_{l,k} \right\}_{k \in K}, P_m = s[r_1, r_2] ! !_n \langle \tilde{e} \rangle . Q,$$
and $P_o = s[r_2, r_1] ? \{ \! \left\langle \! \left\langle \tilde{x}_k \right\rangle \! \right\rangle . Q_k \}_{k \in K'}$
7. (a) $\text{SGP}'\left(\{P_i\}_{i \in I_1}, \{(G_j, s_j)\}_{j \in J_1}\right) \parallel \text{SGP}'\left(\{P_i\}_{i \in I_2}, \{(G_j, s_j)\}_{j \in J_2}\right)$,
else if there are some $I_1 \cup I_2 = I, J_1 \cup J_2 = J$ such that $J_1 \cap J_2 = \emptyset$ and

$$\bigcup_{i \in I_k} \text{act}(P_i) = \{s_j[r] \mid j \in J_k \wedge r \in r(G_j)\}$$
 for $k \in \{1, 2\}$
- (b) $\text{SGP}'\left(\{P_i\}_{i \in I}, \{(G_{k1}, s_k), (G_{k2}, s_k)\} \cup \{(G_j, s_j)\}_{j \in J}\right)$,
else if there is $k \in J$ such that $G_k = G_{k1}, G_{k2}$
8. $(\mu X_G) \text{SGP}'\left(\{P_i\}_{i \in I}, \{(G'_j, s_j)\}_{j \in J}\right)$,
else if $G_j = (\mu t_j) G'_j$ for all $j \in J$ and $\mathcal{G} = \{t_j\}_{j \in J}$
9. $\tau. \text{SGP}'\left(\{P'_1, \dots, P'_n\} \cup \{P_i\}_{i \in I \setminus \{k1, \dots, kn\}}, \{(G_j, s_j)\}_{j \in J}\right)$,
else if there are $k1, \dots, kn \in I$ such that $P_{K1} = \bar{a}[2..n](s).P'_1,$
 $P_{k2} = a(s[2]).P'_2, \dots, P_{kn} = a(s[n]).P'_n$
10. if c then $\text{SGP}'(\{P_{k1}\} \cup \mathcal{P}, \mathcal{G})$ else $\text{SGP}'(\{P_{k2}\} \cup \mathcal{P}, \mathcal{G})$
with $\mathcal{P} = \{P_i\}_{i \in I \setminus \{k\}}$ and $\mathcal{G} = \{(G_j, s_j)\}_{j \in J}$,
else if there is $k \in I$ such that $P_k = \text{if } c \text{ then } P_{k1} \text{ else } P_{k2}$

To deal with multiple sessions (and their global types), we split Case 1 into a case to introduce the SGP-process $\mathbf{0}$ as soon as the set of considered global types is empty (Case 1a) and a case to remove terminated global types end from the set $\{G_j\}_{j \in J}$ (Case 1b). In a similar way, we split Case 7 into a case to introduce a parallel composition of SGP-processes if the considered sets of processes can be partitioned into two sets that implement the actors of different sessions (Case 7a) and a case to split parallel global types (Case 7b), i. e., to replace $\{G_j\}_{j \in J}$ by $\{G_{k1}, G_{k2}\} \cup \{G_j\}_{j \in J \setminus \{k\}}$ if there is a $k \in J$ such that $G_k = G_{k1}, G_{k2}$. The adaptation of the Cases 3, 4, 5, 6, 9, and 10 to multiple global types is straightforward. The Cases 2 and 8 for recursion, are replaced by variants that require all types of the considered set of global types to be reduced to a type variable or a recursive global type, respectively. With that we follow [3], that similarly requires that interleaved sessions can be joined in recursion. The remaining cases are straightforwardly adapted to sets of types.

Note that an implementation of this algorithm should exploit the acyclic dependency relation that is build according to [3] between interactions of different sessions. This relation tells us for the Cases 6 and 9, whether the required communication partners for a session are already unguarded or guarded by another session. In the latter case this communication will introduce a dependency from another session to this session and the respective case cannot be applied. Similarly, this relation tells us for Case 7a that it is possible to introduce a SGP parallel composition if and only if we can split the set of sessions into two disjoint sets such that there are no dependencies between the sessions in different sets.

We overload the definition of $\text{SGP}^*(\cdot, \cdot)$ for interleaved sessions. Let P be well-typed w. r. t. $\{(G_j, s_j)\}_{j \in J}$ and $S = \text{SGP}'(\{P\}, \{(G_j, s_j)\}_{j \in J})$. Then the corresponding SGP-system is $\text{SGP}^*(\{P\}, \{(G_j, s_j)\}_{j \in J}) = \langle \mathcal{V}; S \rangle$, where \mathcal{V} is the vector of names in S .

Note that the results of Sect. 4 are proved in [26] for both variants: $\text{SGP}(\cdot, \cdot)$ and $\text{SGP}'(\cdot, \cdot)$. As we claim, we can extend this algorithm to all variants of MPST that satisfy linearity, i. e., all MPST variants we are aware of. This also includes variants with session delegation. Delegation can be handled similarly to session invitations using a substitution for the delegated session name.

4 Relating the Implementation and Its Sequentialisation

We show that for all processes P that are well-typed w. r. t. the global types $\{(G_j, s_j)\}_{j \in J}$, the mapping $\text{SGP}(\{P\}, \{(G_j, s_j)\}_{j \in J})$ is defined and returns a SGP-process. Therefore, we show that all cases of Definition 3 except for Case 8 preserve well-typedness in their recursive calls. By an induction over J and the structure of the respective types, we show then that—after some preparation steps in the Cases 3, 4, 5, and 10 that do not alter the type—well-typedness ensures that the structure of the system is as required by the respective case to reduce the types. The case of a single session—if P is well-typed w. r. t. G then $\text{SGP}(\{P\}, G)$ is a SGP-process—is a special case of the following theorem.

Theorem 2. *If the process P is well-typed w. r. t. $\{(G_j, s_j)\}_{j \in J}$ then $\text{SGP}^*(\{P\}, \{(G_j, s_j)\}_{j \in J})$ is defined and returns a SGP-process.*

Given a well-typed process, the computation of the mapping and the size of the constructed SGP-process are usually linear in the size of P combined with the sum of the sizes of its types. As discussed in [26], an exponential blow-up cannot be completely avoided but results only from not optimal conditionals, i. e., conditionals that are not only used to branch between alternative labels of an immediately following sender, or from actors that are not influenced by the choice of a branch of a communication in that the sender is preceded by such a conditional. By design, the algorithm in Definition 2 will even in these bad cases minimize the size of the generated SGP-process by delaying the mapping of conditionals as long as possible. In general the computation of the SGP-system is efficient, i. e., usually fast, and the construction does not suffer from the problem of state space explosion, i. e., the generated SGP-system is usually not considerably larger than the original system. Since the construction sequentialises the original system and thereby removes all forms of interaction and restriction, the verification of the SGP-abstraction is much easier than the verification of the original system.

It remains to show, that the SGP-abstraction of a well-typed system that is generated by $\text{SGP}^*(\cdot, \cdot)$ and the original system are semantically similar enough, such that the analysis of the SGP-abstraction allows to verify properties of the original system. Intuitively, a well-typed system and its sequentialisation into a SGP-system have the same steps, but SGP-systems may force an order on steps that are unordered in the original system. This happens for global types such as $1 \rightarrow 2 : !\langle \mathbb{N} \rangle . 3 \rightarrow 4 : !\langle \mathbb{N} \rangle . \text{end}$ that combine causally unrelated communications sequentially.

Example 1. Consider the global type $G = 1 \rightarrow 2 : !\langle \mathbb{N} \rangle . 3 \rightarrow 4 : !\langle \mathbb{N} \rangle . \text{end}$ that consists of two causally independent communications. The system

$$P = \bar{a}[2..4](s).s[1, 2]! \langle 5 \rangle . \mathbf{0} \mid a(s[2]).s[2, 1]?l(x). \mathbf{0} \\ \mid a(s[3]).s[3, 4]! \langle 4 \rangle . \mathbf{0} \mid a(s[4]).s[3, 4]?l(x). \mathbf{0}$$

is a well-typed implementation of this global type. The algorithm of Definition 2 maps this process to the SGP-system $\text{SGP}^*(P, G) = \langle (x_2, x_4); S \rangle$, where $S = \tau.x_2 := 5.x_4 := 4. \mathbf{0}$. The process P has, modulo structural congruence, two maximal runs

$$P \mapsto P' \begin{cases} \rightarrow (\nu s)(s[3, 4]! \langle 4 \rangle . \mathbf{0} \mid s[3, 4]?l(x). \mathbf{0}) \\ \rightarrow (\nu s)(s[1, 2]! \langle 5 \rangle . \mathbf{0} \mid s[2, 1]?l(x). \mathbf{0}) \end{cases} \rightarrow \mathbf{0}$$

where $P' = (\nu s)(s[1, 2]! \langle 5 \rangle . \mathbf{0} \mid s[2, 1]?l(x). \mathbf{0} \mid s[3, 4]! \langle 4 \rangle . \mathbf{0} \mid s[3, 4]?l(x). \mathbf{0})$. But the abstraction $\text{SGP}^*(P, G)$ simulates only the sequence of steps at the top

$$\langle (x_2 = 0, x_4 = 0); S \rangle \mapsto \langle (x_2 = 0, x_4 = 0); x_2 := 5.x_4 := 4. \mathbf{0} \rangle \\ \mapsto \langle (x_2 = 5, x_4 = 0); x_4 := 4. \mathbf{0} \rangle \mapsto \langle (x_2 = 5, x_4 = 4); \mathbf{0} \rangle$$

in that first process 2 receives the value 5—and the SGP-process accordingly updates the variable x_2 of 2—and then 4 receives the value 4.

Except for such sequentialisations from the global type, the original system and its SGP-system are similar. In particular, this means that each step of the SGP-system can be simulated by the original system. Thus, SGP-systems do not introduce new behaviour.

Theorem 3 (Soundness). *If P is well-typed w. r. t. \mathcal{G} then for all \mathcal{S}' such that $\text{SGP}^*(\{P\}, \mathcal{G}) \mapsto \mathcal{S}'$ there exist some P', \mathcal{G}' such that $P \mapsto P'$, P' is well-typed w. r. t. \mathcal{G}' , and $\text{SGP}^*(\{P'\}, \mathcal{G}') \equiv_{\mathcal{S}} \mathcal{S}'$.*

Although the SGP-system can perform intuitively the same steps as the original system, the order that is forced on some steps by the above discussed sequentialisations prevents us from obtaining the same result in the other direction. However, only the order of steps can differ. Because of that, whenever the original system does a step there is a sequence of steps bringing the original system towards a state that can be reached by the SGP-system. To prove this result, we rely on the observation that the behaviour of a SGP-process follows the global type it was constructed from and well-typedness forces processes to similarly follow the specification in their types. Since renamings of input binders change the vector of variables of a SGP-system, we assume that no alpha conversion is used to rename input binders in the following.

Theorem 4 (Completeness). *Let $\mathcal{G} = \{(G_j, s_j)\}_{j \in J}$. If the system P is well-typed w. r. t. \mathcal{G} then for all $P \mapsto P'$ there exist P'', \mathcal{G}'' such that $P' \mapsto^* P''$, P'' is well-typed w. r. t. \mathcal{G}'' , and $\text{SGP}^*(\{P\}, \mathcal{G}) \mapsto^* \text{SGP}^*(\{P''\}, \mathcal{G}'')$.*

Interestingly, the combination of Theorem 3 and Theorem 4 is similar to (*weak*) *operational correspondence* as it is introduced in [14] as criterion for the quality of encodings. Using the results from [25], then the sequentialisation of a system is correspondence similar to the original system, where correspondence simulation \lesssim was introduced in [25].

Corollary 1. *If P is well-typed w. r. t. $\mathcal{G} = \{(G_j, s_j)\}_{j \in J}$ then $\text{SGP}^*(P, \mathcal{G}) \lesssim P$.*

Correspondence similarity is strictly weaker than bisimilarity, but it implies *coupled similarity*. Coupled similarity was introduced in [23] as a weaker variant of bisimilarity that allows to relate the distributed implementation to a global specification. Similarly, we relate the sequentialisation $\text{SGP}^*(P, \mathcal{G})$ to the distributed implementation in P . As explained in [23], bisimilarity is in general too strict to relate a distributed implementation with a global specification. So, following the hierarchy in [12, 13], coupled similarity (or the only slightly stronger correspondence simulation) is intuitively the strictest simulation relation that we could expect here.

5 Using SGP-Abstractions for Verification

To verify properties that are based on data values, we can use standard verification techniques such as model checking on the generated SGP-systems. The correspondence simulation $\text{SGP}^*(P, G) \lesssim P$ between the SGP-system and the original system P ensures that properties that are valid for the SGP-system are also valid for P , if these properties are preserved modulo \lesssim . For the presented approach, this is the case for all properties on the values of data variables—that reflect the reception of values in the original system—that do not require to compare different such variables that are updated concurrently in the original system as explained in Example 1.

Accordingly, we cannot use this method to verify properties on the relation between variables that are updated concurrently in the original system. This is because, we use the structure of the global type to sequentialise. If—as in the case of G in Example 1—the global type combines independent communications sequentially, then the mapping $\text{SGP}(P, \mathcal{G})$ forces an order on the corresponding value updates following the global types \mathcal{G} and not the process P .

However, this problem occurs only w. r. t. communications that are causally unrelated, i. e., such properties are in general problematic in distributed systems. Since these values are altered independently in the original system, properties that relate their values will often not hold in all runs. The easiest way to avoid such false positives, is to compute the causal relation of the communications in \mathcal{G} . Remember that global types do not contain binders. Thus, computing the causality relation for a single session can be done in a similar way as in the π -calculus (compare e. g. to [27]), but does not have to bother about binders and scope extrusion. To obtain a causality relation for the case of globally interleaved sessions, we combine the relation that captures dependencies between the interactions of different sessions of [3] with the causality within a session. A property of the SGP-system then holds for the original system if it is invariant under different linearisations of this causal order.

To illustrate the verification of system properties, we use the model checker *Spin* [16, 17] and translate SGP-systems into *Promela*, the input language of *Spin*. Therefore, we provide an algorithm to translate a SGP-process into *Promela* code. This algorithm serves as a role model to obtain similar mappings for other model checkers. We choose *SPIN* to illustrate how our algorithm for well-typed systems compares to the standard partial order reduction techniques that are implemented in *SPIN* and that work without the type information. Other implementations might prefer a model checker that is specialised on the analyses of data instead of concurrency issues such as *NUXMV* [8].

First we generate a preamble for the *Promela* program, i. e., declare variables and set their initial values. The variables are obtained from the vector of variables \mathcal{V} in a SGP-system $\langle \mathcal{V}; S \rangle$. Sometimes the initial values are directly specified by the implementation or are given as parameters of the implementation. Otherwise, the developer has to pick suitable initial values respecting their respective sorts.

The translation into *Promela* consists of two layers. The outer layer $\llbracket S \rrbracket$ creates the proctype that is required by *Promela* and passes the term onto the inner

layer $\llbracket S \rrbracket_i$. The inner layer is simple: Variable assignments are translated to variable assignments encapsulated in an atomic block if multiple assignments are done simultaneously. Recursion is implemented by introducing a label and jumping to that label when the recursion variable is called.

Definition 4 (Translation of SGP-Processes into Promela)

$$\begin{aligned} \llbracket S \rrbracket &= \text{active proctype ModelS()} \{ \\ &\quad \llbracket S \rrbracket_i \\ &\quad \text{LEnd;} \\ &\} \\ \llbracket \tilde{v} := \tilde{e}.S \rrbracket_i &= \text{atomic} \{v_1 = e_1; \dots; v_n = e_n; \} \llbracket S \rrbracket_i \\ \llbracket \text{if } c \text{ then } S_1 \text{ else } S_2 \rrbracket_i &= \text{if} \\ &\quad :: c \quad \rightarrow \llbracket S_1 \rrbracket_i \\ &\quad :: \text{else} \rightarrow \llbracket S_2 \rrbracket_i \\ &\quad \text{fi} \\ \llbracket S_1 \parallel S_2 \rrbracket_i &= \text{run}(S_1); \text{run}(S_2) \\ \llbracket \mathbf{0} \rrbracket_i &= \text{goto LEnd;} \\ \llbracket \tau.S \rrbracket_i &= \text{skip}; \llbracket S \rrbracket_i \\ \llbracket (\mu X) S \rrbracket_i &= \text{LX}; \llbracket S \rrbracket_i \\ \llbracket X \rrbracket_i &= \text{goto LX;} \end{aligned}$$

where for each $\text{run}(S_i)$ a separate **proctype** is introduced with $\llbracket S_i \rrbracket$.

Note that, if \tilde{v} and \tilde{e} are singletons in $\tilde{v} := \tilde{e}.S$, the atomic block is omitted.

In [26] we present a toy example to illustrate our approach. It implements a simple auctioneer system consisting of an auctioneer and two bidders. After translating this example into a SGP-system and then into **Promela**, some properties given as LTL-formulae are verified in **SPIN**.

Moreover, to visualise the state-space explosion problem, we implemented the key-exchange part of the Needham-Schroeder protocol. We derived the sequentialised (s) version out of the distributed (d) version using our algorithm. The following table shows time and memory needed to check our **Promela** implementation of the Needham-Schroeder protocol. **Spin** crashed before it could compute the distributed versions for more than 6 participants.

participants	2(d)	2(s)	4(d)	4(s)	6(d)	6(s)	8(d)	8(s)	10(d)	10(s)
seconds	0.01	0	0.19	0	51.7	0.05	–	1.27	–	36.2
MB	128	128	137	129	1836	138	–	360	–	5809

6 Conclusions

We introduce a mapping from well-typed systems—that are distributed concurrent systems that interact by communication—into SGP-systems—that simulate the information flow of the original system from a global point of view and that do not have interactions. The algorithm to compute these SGP-systems is usually linear in the size of the original system. Without interactions and only finite vectors of variables, the verification of properties is significantly more efficient for SGP-systems than for the original system. The presented mapping ensures that properties that hold for the SGP-system are also valid for the original system modulo coupled similarity. Finally, we present a second mapping from SGP-systems into *Promela*; the input language of the *Spin* model checker.

To formalise the relation between the original system and its sequentialisation into a SGP-system, we relate them by correspondence simulation. Correspondence simulation was described in [25] to describe the relation that the criterion operational correspondence forces on processes and their encodings. As discussed in [14, 15], operational correspondence is essential to reason about the quality of encodings between process calculi. In this sense, the presented mapping is a good encoding. Moreover, correspondence similarity implies coupled similarity. As discussed in [23], coupled similarity is a good way to relate central specifications—or in our case global sequentialisations—to their distributed implementations. Since bisimilarity is in general too strict to relate the original system and its sequentialisation, coupled similarity (or the only slightly stronger correspondence simulation) is intuitively the strictest simulation relation that we could expect here.

Multiparty session types are already a very efficient verification tool for all properties about the communication structure of systems. The presented sequentialisation allows us to benefit from their efficiency also in the verification of properties that are usually not in the range of type systems, because they require the consideration of concrete runs of the system or the values of variables.

Due to the interleaving of independent actions, the state space of a concurrent system is in the worst case exponentially larger than of its sequentialisation. As an example, we implemented the *Needham-Schroeder public key protocol* with 10 pairs of processes that interact with the same server (see [26]). *Spin* generated for the original system more than 35 million states (matching more than 154 million states while using more than 7,5GB memory) before crashing after 969 seconds. For the sequentialisation *Spin* computed the complete model in only 62 seconds generating 75 million states.

The *Scribble* project [18, 31] provides a tool set that allows to specify and check multiparty session types. They also provide a tool to check a given implementation against a given type. The presented algorithms could support such tool sets by increasing the kinds of properties that can be analysed within such a tool set, while the efficiency of such tools is not negatively influenced. In fact, the derivation of SGP-abstractions can be directly integrated into the type check such that SGP-abstractions are produced as a by-product of type checking.

References

1. Bakst, A., Gleissenthall, K.V., Kıcı, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. In: Proceedings of the ACM on Programming Languages 1(OOPSLA), 110:1–110:27 (2017). <https://doi.org/10.1145/3133934>
2. Bejleri, A., Yoshida, N.: Synchronous multiparty session types. *Electron. Notes Theor. Comput. Sci.* **241**, 3–33 (2009). <https://doi.org/10.1016/j.entcs.2009.06.002>
3. Bettini, L., Coppo, M., D’Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_33
4. Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE-2013. LNCS, vol. 7892, pp. 50–65. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38592-6_5
5. 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. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_12
6. Caires, L., Pérez, J.A.: Multiparty session types within a canonical binary theory, and beyond. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 74–95. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_6
7. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. *Distrib. Comput.* **31**(1), 51–67 (2018). <https://doi.org/10.1007/s00446-017-0295-1>
8. Cavada, R., et al.: The nUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
9. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst. (TOCS)* **3**(1), 63–75 (1985). <https://doi.org/10.1145/214451.214456>
10. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 424–440. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_25
11. Demangeon, R., Honda, K.: Nested protocols in session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 272–286. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_20
12. Glabbeek, R.J.: The linear time — branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_6
13. van Glabbeek, R.: The linear time - branching time spectrum i: the semantics of concrete, sequential processes. *Handbook of Process Algebra*, pp. 3–99 (2001)
14. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.* **208**(9), 1031–1053 (2010). <https://doi.org/10.1016/j.ic.2010.05.002>
15. Gorla, D., Nestmann, U.: Full abstraction for expressiveness: history, myths and facts. *Math. Struct. Comput. Sci.* **26**(4), 639–654 (2014). <https://doi.org/10.1017/S0960129514000279>
16. Holzmann, G.J.: *Design and Validation of Computer Protocols*. Prentice Hall, Upper Saddle River (1991)

17. Holzmann, G.J.: The model checker SPIN. *IEEE Trans. Software Eng.* **23**(5), 279–295 (1997). <https://doi.org/10.1109/32.588521>
18. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) *ICDCIT 2011*. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19056-8_4
19. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: *Proceedings of POPL*, vol. 43, pp. 273–284. ACM (2008). <https://doi.org/10.1145/1328438.1328472>
20. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. *J. ACM (JACM)* **63**(1) (2016). <https://doi.org/10.1145/2827695>
21. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. *Inf. Comput.* **100**(1), 1–77 (1992). [https://doi.org/10.1016/0890-5401\(92\)90008-4](https://doi.org/10.1016/0890-5401(92)90008-4)
22. Montesi, F.: *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen (2013). http://www.fabriziomontesi.com/files/choreographic_programming.pdf
23. Parrow, J., Sjödin, P.: Multiway synchronization verified with coupled simulation. In: Cleaveland, W.R. (ed.) *CONCUR 1992*. LNCS, vol. 630, pp. 518–533. Springer, Heidelberg (1992). <https://doi.org/10.1007/BFb0084813>
24. Peled, D.: Ten years of partial order reduction. In: Hu, A.J., Vardi, M.Y. (eds.) *CAV 1998*. LNCS, vol. 1427, pp. 17–28. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0028727>
25. Peters, K., van Glabbeek, R.: Analysing and comparing encodability criteria. In: *Proceedings of EXPRESS/SOS, EPTCS*, vol. 190, pp. 46–60 (2015). <https://doi.org/10.4204/EPTCS.190.4>
26. Peters, K., Wagner, C., Nestmann, U.: taming concurrency for verification using multiparty session types (Technical Report). Technical report, TU Berlin/TU Darmstadt, <https://arxiv.org> (2019)
27. Priami, C.: *Enhanced Operational Semantics for Concurrency*. Ph.D. thesis, Università di Pisa-Genova-Udine, August 1996
28. Toninho, B., Yoshida, N.: Certifying data in multiparty session types. *J. Logical Algebraic Methods Program.* **90**, 61–83 (2017). <https://doi.org/10.1016/j.jlamp.2016.11.005>
29. Wagner, C., Nestmann, U.: States in process calculi. In: *Proceedings of EXPRESS/SOS, EPTCS*, vol. 160, pp. 48–62 (2014). <https://doi.org/10.4204/EPTCS.160.6>
30. Yoshida, N., Deniérou, P.-M., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: Ong, L. (ed.) *FoSSaCS 2010*. LNCS, vol. 6014, pp. 128–145. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12032-9_10
31. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) *TGC 2013*. LNCS, vol. 8358, pp. 22–41. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05119-2_3