A SYMMETRIC PROTOCOL TO ESTABLISH
SERVICE LEVEL AGREEMENTS

JAN FRISO GROOTE AND TIM A.C. WILLEMSE

Department of Mathematics and Computer Science, Eindhoven University of Technology
e-mail address: {J.F.Groote,T.A.C.Willemse}@tue.nl
URL: www.win.tue.nl/~{jfg,timw}

ABSTRACT. We present a symmetrical protocol to repeatedly negotiate a desired service level between two parties, where the service levels are taken from some totally ordered finite domain. The agreed service level is selected from levels dynamically proposed by both parties and parties can only decrease the desired service level during a negotiation. The correctness of the protocol is stated using modal formulas and its behaviour is explained using behavioural reductions of the external behaviour modulo weak trace equivalence and divergence-preserving branching bisimulation. Our protocol originates from an industrial use case and it turned out to be remarkably tricky to design correctly.

1. INTRODUCTION

We consider the problem of repeatedly establishing a service level agreement between two parties using a symmetric protocol (*i.e.*, one in which both parties use the same algorithm), without resorting to a centralised decision making process. The set of service levels is some totally ordered subrange of the natural numbers. The involved parties independently and dynamically propose desired service levels. When the proposed service levels match, the protocol must inform both parties which service level is agreed upon. Once a party proposes a service level this puts an upper bound on it: proposals for lower service levels will still be taken into account but any subsequent proposal to increase the service level will be ignored.

The problem finds its origin at ASML (www.asml.com), a producer of lithography machines. In their setting, two (distributed) parties need to determine whether to swap or flush two chucks holding a silicon wafer. When both parties propose to swap the chucks, swapping is required. If one party decides to flush, *e.g.*, when a wafer is damaged, the other party has to concede, resulting in a flush. The swap and flush decisions can be viewed as service levels, with flush the lower of the two service levels. It turns out it is annoyingly hard to correctly design a symmetric protocol that solves the problem, symmetry and the absence of a centralised decision making process being part of ASML's initial design requirements for the protocol: using modal formulas and model checking, all proposed solutions were shown to be incorrect. Since allowing for a centralised decision making process significantly simplifies

the problem, ASML ultimately adopted such a protocol; this solution was subsequently completely formally modelled and verified [NRG19].

This leaves an open question, *viz.*, whether it is at all possible to have a symmetric protocol to establish a service level agreement between two distributed parties. Motivated by the many failed attempts we set out to prove that no symmetric protocol solving the problem exists. Much to our surprise, the proof left one scenario open, yielding conditions for a working negotiation scheme. Generalising the resulting protocol to arbitrary sets of (ordered) service levels, which has applications in decentralised decision making beyond ASML's restricted setting, turns out to be reasonably straightforward. We therefore present the more generic repeated service level agreement negotiation protocol for two parties and arbitrary (non-empty) sets of service levels.

The next question is how to provide a convincing argument that our protocol is actually correct. This is not trivial, because the protocol is continuously interacting with its environment. As a result, Hoare logics, contracts or other proof techniques for non-reactive systems are not applicable. Moreover, since the decision making cannot be centralised in a symmetric solution, the two negotiating parties are not necessarily in the same round at every point in time. This makes it improbable that a solution has an elegant, simple overall external behaviour, implying that it is not feasible to phrase a natural specification of the external behaviour of the protocol and proving conformance of our protocol to that specification.

The only technique that we found suitable to claim the correctness of our solution is to show, by means of model checking, that our formalisation of the protocol meets a number of desirable modal formulas. These formulas, however, are non-trivial. Therefore, in addition to verifying the correctness of our protocol, we illustrate aspects of its external behaviour, reduced with respect to weak trace equivalence and divergence-preserving branching bisimulation. While these pictures are quite insightful for understanding the protocol, they can be quite misleading and cannot be used as correctness arguments; *e.g.*, weak trace equivalence does not preserve the branching structure of a system. Moreover, weak trace equivalence masks deadlocks; in fact, we have seen several solutions for which the weak trace equivalence reduced models look perfectly reasonable, but which suffer from deadlocks.

Apart from serving to prove the correctness of our solution, the modal formulas turned out to be instrumental in the process of designing our protocol. Model checking the formulas typically quickly revealed logical flaws in variations of our protocol, and the counter-examples [WW18, CLW15] helped to pinpoint the underlying reasons and steered the process of making the necessary amendments. It is without a doubt that the use of model checking has greatly accelerated the process of designing our protocol.
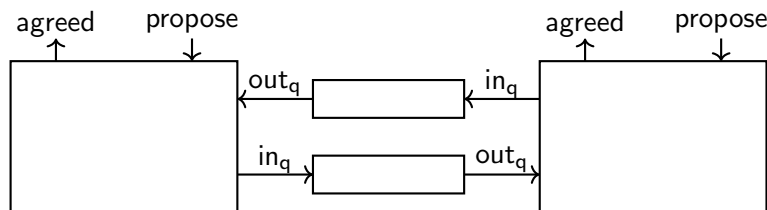

Figure 1: The structure of two processes negotiating an agreement for quality of service.

Related Work. There are various protocols for establishing service level agreements in a variety of domains, see *e.g.* [OTM$^+$07, BdM06, PKB06]. Most of these protocols focus on asymmetric solutions. Moreover, little effort seems to have been given to formally verifying such protocols. We here focus on the few works we are aware of, and that do use formal verification.

In [CKML13], Chalouf *et al.* study a protocol for service level negotiation, covering both quality of service and security. With the help of formal verification using the SPIN model checker, the authors uncover various deadlocks and propose solutions to fix these deadlocks. Unlike our protocol, which treats both parties using the protocol symmetrically, their protocol is based on the *request-response* model, in which one party initiates negotiation and the other party responds.

The algorithm studied in [DDO06], which is again asymmetric, is a bargaining algorithm in which sellers and buyers are to reach an agreement through a process of 'offer/counter-offer'. The algorithm implements a naive negotiation scheme and terminates upon establishing agreement or when one party withdraws from the negotiation; repeatedly establishing agreement through the same algorithm is not supported. A formal SPIN model is used to simulate the algorithm.

Finally, in [YYWL12], Yaqub *et al.* describe a protocol development framework for service level agreement negotiations in cloud and service-based systems. They illustrate the modelling and verification phase of their framework by developing a multi-tier, multi-round, customisable negotiation protocol called the *Simple Bilateral Negotiation Protocol*. Also in this case, the protocol is formalised in SPIN. The authors abstract from the service levels that can be sent and use a take-it-or-leave-it interaction scheme. Their formalisation reveals various deadlock and livelock scenarios in early designs of the protocol. The final model is subjected to a number of LTL properties, focussing mainly on reaching and/or transitioning various phases of the protocol.

We remark that the problem of establishing service level agreements is conceptually close to the (distributed) consensus problem. Both revolve around reaching agreement on a proposal, but in our case, the proposal can change dynamically in a round; we are not aware of a variation in the distributed consensus problem with similar assumptions. Distributed consensus is studied for, *e.g.*, systems that communicate (partly) synchronously and asynchronously, and under various types of failures. In [FLP85], it is shown that there is no deterministic algorithm to solve the distributed consensus problem in an asynchronous message passing setting, in case there is at least one failing process; we expect that this result extends to our setting when generalising our problem to multiple processes which are allowed to fail.

Outline. In the subsequent sections we present the protocol in the process specification language mCRL2 (Section 3) and explain its external behaviour, reduced with respect to weak trace equivalence and divergence-preserving branching bisimulation in Section 4. In Section 5, we formulate the requirements as modal formulas and verify them for increasing ranges of qualities of service. In Section 6, we state some conclusions and future work. We first phrase the problem statement and the requirements the protocol must satisfy in Section 2.
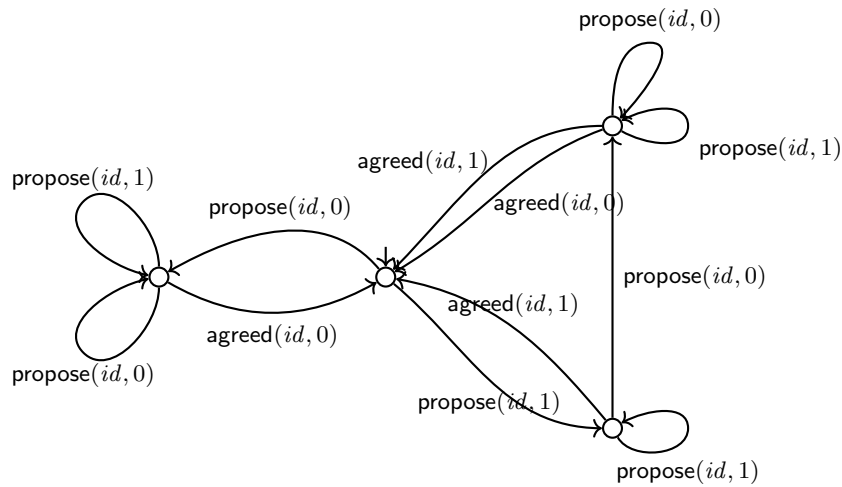
Figure 2: The external behaviour for one single party id with two qualities of service
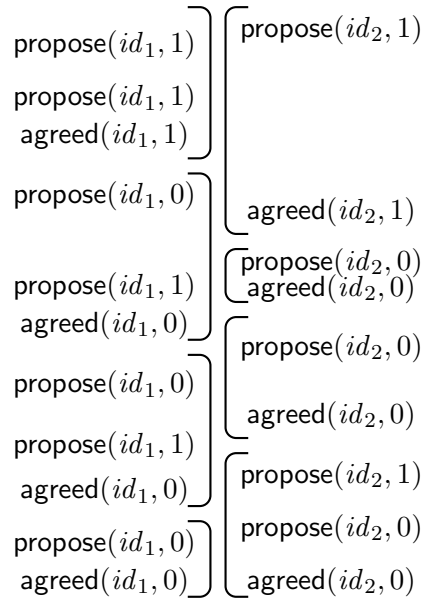

Figure 3: Four rounds of the protocol

2. REQUIREMENTS FOR A SERVICE LEVEL AGREEMENT NEGOTIATION PROTOCOL

The problem we address is that of repeatedly, in *rounds*, negotiating a service level agreement between two parties (identified by id_1 and id_2, respectively), where (1) the decision making process is decentralised and (2) both parties use an instance of the same algorithm, which is oblivious to which party is running it. Communication between the two parties proceeds asynchronously via message passing along reliable channels, and the set of service levels is a totally ordered subrange of the natural numbers. The required architecture is depicted in Figure 1.

In each round, both parties aim to establish a common service level. Service levels can be proposed using the action $\mathsf{propose}(id, l)$ where id is the identifier of this party and l is a service level. A party may repeatedly propose service levels and the protocol is to process all these proposals. Once a party proposes a service level, subsequent proposals for higher service levels must be ignored but lower service levels may still be agreed upon. A round ends for this party when the action $\mathsf{agreed}(id, l')$ takes place. The party can then start the negotiation for a new common service level in this new round. The service levels proposed in one round are not to affect the decision on a common service level in a later round.

From the perspective of a single party id, a solution to the problem must meet the following requirements on the agreed quality of service at the end of a round:

(I) The agreed quality of service l' must have been proposed by party id in the same round.
(II) If at some time in a round a service level l is proposed by party id, a higher service level l' proposed only after l will never be agreed in this round.

It is allowed that a party successively proposes different services levels l and l' with $l' < l$. In this case the protocol can agree on both service levels. Figure 2 depicts a labelled transition system that represents the allowed sequences of proposals and agreements that can occur from the perspective of one process, if there are only two service levels to choose from. This figure nicely illustrates that when a proposal for level 1 is followed by a proposal to settle on level 0, both can be agreed, as after doing the actions $\mathsf{propose}(id, 1)$ followed by $\mathsf{propose}(id, 0)$ from the initial state, it is possible to do both the actions $\mathsf{agreed}(id, 0)$ and $\mathsf{agreed}(id, 1)$. But if initially service level 0 is proposed by doing action $\mathsf{propose}(id, 0)$ in the initial state, service level 0 is the only service level that can be agreed upon, as $\mathsf{agreed}(id, 0)$ is the only agreed action after this proposal.

If we consider two parties simultaneously, we observe that their perspectives on when rounds start and end may not be the same. Figure 3 illustrates such a scenario. We therefore speak of round n for a particular party and not for the protocol as a whole. Round 1 for party id is the time period from the start of the protocol up to and including the time of the first action $\mathsf{agreed}(id, l)$. Round $n+1$ for party id is the time period just after the nth occurrence of an action $\mathsf{agreed}(id, l')$ up to and including the $n+1^{th}$ occurrence of $\mathsf{agreed}(id, l'')$.

From the perspective of both parties, the solution provided by the protocol should meet the following two requirements on the quality of service both parties agree upon:

(III) If during round n for party id_1, action $\mathsf{agreed}(id_1, l)$ occurs and during round n for party id_2 action $\mathsf{agreed}(id_2, l')$ happens, then $l = l'$.
(IV) Whenever during round n both parties propose a common service level l through actions $\mathsf{propose}(id_1, l)$ and $\mathsf{propose}(id_2, l)$, then both parties will reach an agreement, indicated by $\mathsf{agreed}(id_1, l')$ and $\mathsf{agreed}(id_2, l')$ for some l' possibly different from l. The execution of the agreement action can be postponed indefinitely only by the parties repeatedly submitting redundant proposals, or by failing to propose the essential matching $\mathsf{propose}(id, l')$, with l' the minimal service level proposed by the other party and not party id itself.

In case of requirement (IV) there are two typical example traces in which an action agreed does not need to follow within a finite number of steps. The first example scenario is one in which a process executes a $\mathsf{propose}(id_1, 1)$ action and subsequently the other process repeatedly executes action $\mathsf{propose}(id_2, 1)$ (all of which get ignored, except for the first one). This effectively prevents the protocol to report an action $\mathsf{agreed}(id_1, 1)$ and $\mathsf{agreed}(id_2, 1)$, although the protocol can in principle do these.

The second one is process id_1 — probably repeatedly — executes action $\mathsf{propose}(id_1, 1)$ and process id_2 executes action $\mathsf{propose}(id_1, 0)$. As they propose different service levels, an agreement never follows. The protocol ends up in the middle topmost state in Figure 7 (note that in this figure self-loops in states have been omitted for readability).

3. FORMAL MODEL IN MCRL2

In this section we give a full model of our protocol in the process specification language mCRL2 [GM14]. Somewhat to our surprise the protocol is rather concise, especially because it also includes the description of the unbounded channels used by both parties to exchange messages. Being a formal description, it is very straightforward to translate it to any appropriate implementation language.

3.1. **A note on mCRL2.** A detailed description of the specification language mCRL2 is available in the standard textbook [GM14]. To ensure that our presentation of the protocol is self-contained, we here briefly review the operators used in our specification. *Actions* are the basic building blocks of a specification. In our protocol, the actions typically represent activities such as sending or receiving messages. Using the binary sequential composition operator $\cdot$, the binary non-deterministic choice operator $+$ and recursion, processes can be built compositionally. The binary non-deterministic choice is generalised by the quantifier $\sum$, taking a typed variable and a process as arguments. Intuitively, an expression of the form $\sum x{:}T.p$ represents a non-deterministic choice among all possible processes resulting from substituting concrete values, taken from T, for x in p. Recursive specifications can carry typed data parameters and data expressions can be passed as arguments in recursive calls. Finally, the process $b \to p \diamond q$ behaves as p when Boolean expression b evaluates to *true* and q otherwise. Two processes can be composed in parallel using the $\|$ operator. A binary communication operator, taking a *communication function* and a process as arguments, can be used to specify that two (or more) actions that can happen in parallel, synchronise and create a new action. The binary *allow* operator, taking a set of actions A and a process p, can be used to describe the behaviour of p when all actions not in set A are blocked; this operator can be used to 'enforce' synchronisation by only allowing the actions resulting from a communication, and blocking the individual actions involved in the communication.

3.2. **A service level agreement negotiation protocol.** We use natural numbers $\mathbb{N}$ to represent service levels and the standard ordering $\leq$ on natural numbers to model the total order on the service levels, assuming that 0 is the lowest level attainable. The set *Levels*, defined as the set $\{\, l{:}\mathbb{N} \mid l < Max \,\}$, is the set of *valid* levels, where we assume that the value $Max > 0$ is a parameter in our model. The natural number *None*, which we set to *Max*, is used to represent that no service level has been settled yet.

We distinguish between messages used to propose a particular service level, and messages used to communicate that a party decided upon a service level. This is formalised by the data sort *Message*, defined as follows:

$$Message = \mathsf{struct}\ inform(\mathbb{N}) \mid decide(\mathbb{N})$$

The two parties are identified by means of an identifier, taken from a finite collection of identifiers *ID*:

$$ID = \mathsf{struct}\ id_1 \mid id_2$$

Both parties communicate asynchronously via message passing along a uni-directional, reliable channel. The channel can receive messages (via parameterised action $\mathsf{r_q}$) from one party and pass messages (via parameterised action $\mathsf{s_q}$) to the other party. Process Q, depicted below, models the (first-in first-out) channel for party id; the messages sent along the channel are recorded in a queue q of messages, modelled as a list.

$$\mathsf{proc}\ Q(id{:}ID, q{:}List(Message)) =$$
$$\sum_{m:Message} \mathsf{r_q}(id, m) \cdot Q(id, m \triangleright q) + (|q| > 0) \rightarrow \mathsf{s_q}(id, rhead(q)) \cdot Q(id, rtail(q))$$

Remark that $m \triangleright q$, for a list q and element m represents a new list q with m prepended to it. The expression $rhead(q)$, the *rear head*, denotes the last element in q, whereas expression $rtail(q)$ denotes the list without its last element. Note that $head(q)$ and $tail(q)$, used elsewhere, denote the regular head and tail of a queue. The length of list q is denoted by $|q|$.

We next describe the algorithm (the protocol entity) that each of the negotiating parties runs; see Figure 4 for the mCRL2 specification. Each entity uses actions $\mathsf{propose}(id, l)$ and $\mathsf{agreed}(id, l)$ to communicate with its external environment; these actions represent the events explained in the previous section. Communication *between* the two parties proceeds via two instances of the process Q. Each protocol entity can send messages to this queue via action s and receive messages from the queue via action r. The synchronisation between these actions and the actions used by process Q is described by process $SLAN$ at the end of this section.

The protocol entity P (see the first equation in Figure 4) is logically decomposed in two distinct processes: a process P_{other}, handling the messages that arrive from the other party, and a process P_{env} that is triggered by communications with the external environment. Decision making is distributed between these two subprocesses: both proposals for a new service level and messages informing the protocol entity that the other party has proposed a particular service level may lead to decisions being taken. Note that decisions are made by a protocol entity itself, *i.e.*, without any centralised party or manager.

We first focus on the flow of control of process P_{env}, see the second equation in Figure 4. If the two parties have reached a valid decision (which is stored in data parameter *decision*), the protocol entity reverts to a mode in which any newly arriving proposals are ignored until the environment is informed about the agreed service level. As long as no valid decision has been reached, any valid proposal l the entity receives from its environment is processed. In case the new proposal l is strictly lower than those proposed earlier by the environment (stored in parameter *mine*) and not lower than the proposals received so far from the other party (stored in parameter *theirs*), the new proposal l is taken into consideration. In any other case, it is ignored since both parties already communicated a lower service level. Agreement is reached when the considered proposal l agrees with the least proposal so far received from the other party, *i.e.*, when $l \approx theirs$; this is communicated to the other party by informing it of decision l, and setting Boolean *hold*, to indicate that the protocol entity must 'suspend' negotiation until also the other party informs the protocol entity that a decision was reached. If the considered proposal l does not match the lowest service level of the other party, it must decrease the service level considered acceptable for the protocol entity itself. In this case, the other party is informed of this lower service level.

The Boolean *hold* also dictates the control flow in the process that deals with messages received from the other party, see the fifth equation in Figure 4. In case *hold* is high, this

$\mathsf{proc}\ P(id, from{:}ID, mine{:}Set(\mathbb{N}), theirs{:}\mathbb{N}, decision{:}\mathbb{N}, hold{:}\mathbb{B}) =$
$\quad P_{env}(\,) + P_{other}(\,)$

$\mathsf{proc}\ P_{env}(id, from{:}ID, mine{:}Set(\mathbb{N}), theirs{:}\mathbb{N}, decision{:}\mathbb{N}, hold{:}\mathbb{B}) =$
$\quad (decision \approx None)\ \to P_{negotiate}(\,) \diamond P_{decide}(\,)$

$\mathsf{proc}\ P_{decide}(id, from{:}ID, mine{:}Set(\mathbb{N}), theirs{:}\mathbb{N}, decision{:}\mathbb{N}, hold{:}\mathbb{B}) =$
$\quad \mathsf{agreed}(id, decision) \cdot P(decision = None)\ +$
$\quad \sum_{l:\mathbb{N}} (l \in Levels) \to \mathsf{propose}(id, l) \cdot P(\,)$

$\mathsf{proc}\ P_{negotiate}(id, from{:}ID, mine{:}Set(\mathbb{N}), theirs{:}\mathbb{N}, decision{:}\mathbb{N}, hold{:}\mathbb{B}) =$
$\quad \sum_{l:\mathbb{N}} (l \in Levels) \to \mathsf{propose}(id, l)\ \cdot$
$\qquad (l < min(mine) \wedge l \leq theirs)$
$\qquad\quad \to\ ((l \approx theirs)$
$\qquad\qquad\quad \to \mathsf{s}(id, decide(l)) \cdot P(mine = \emptyset, theirs = None, decision = l, hold = true)$
$\qquad\qquad\quad \diamond\ \mathsf{s}(id, inform(l)) \cdot P(mine = mine \cup \{l\})$
$\qquad\quad \diamond\ P(\,))$

$\mathsf{proc}\ P_{other}(id, from{:}ID, mine{:}Set(\mathbb{N}), theirs{:}\mathbb{N}, decision{:}\mathbb{N}, hold{:}\mathbb{B}) =$
$\quad \sum_{l:\mathbb{N}} hold \to\ (\mathsf{r}(from, inform(l)) \cdot P(\,) + \mathsf{r}(from, decide(l)) \cdot P(hold = false))$
$\qquad \diamond\ (\ \mathsf{r}(from, decide(l)) \cdot \mathsf{s}(from, decide(l)) \cdot P(mine = \emptyset, theirs = None, decision = l)\ +$
$\qquad\quad \mathsf{r}(from, inform(l))\cdot$
$\qquad\quad (l \in mine \wedge decision \approx None)$
$\qquad\qquad \to\ \mathsf{s}(id, decide(l)) \cdot P(mine = \emptyset, theirs = None, decision = l, hold = true)$
$\qquad\qquad \diamond\ P(theirs = min(theirs, l))\,)\,)\,)$

Figure 4: Model of the protocol entity negotiating the desired service level agreement.

means that the protocol entity is currently waiting for the other party to confirm that it has reached a decision. If this is the case, *hold* becomes low, but, until this happens, all suggested service levels that the protocol entity receives from the other party are ignored. Whenever negotiation is still in progress, it may be the case that the other party has sent a decision l, to which the protocol entity then replies by confirming this decision and storing the decision in parameter *decision*; all previously proposed service levels, both from the other party and the protocol entity itself, are reset. In case the other party has sent a new service level l which was previously proposed through the protocol entity itself, and no service level has yet been decided upon, l is the service level both parties will agree upon. This is then communicated to the other party. Whenever l was not previously proposed and is below the recorded minimal proposal (stored in *theirs*) received from the other party, the new minimal level is stored.

The service level agreement negotiation protocol, which is the composition of the two protocol entities communicating via the unidirectional channels, see Figure 1, is described
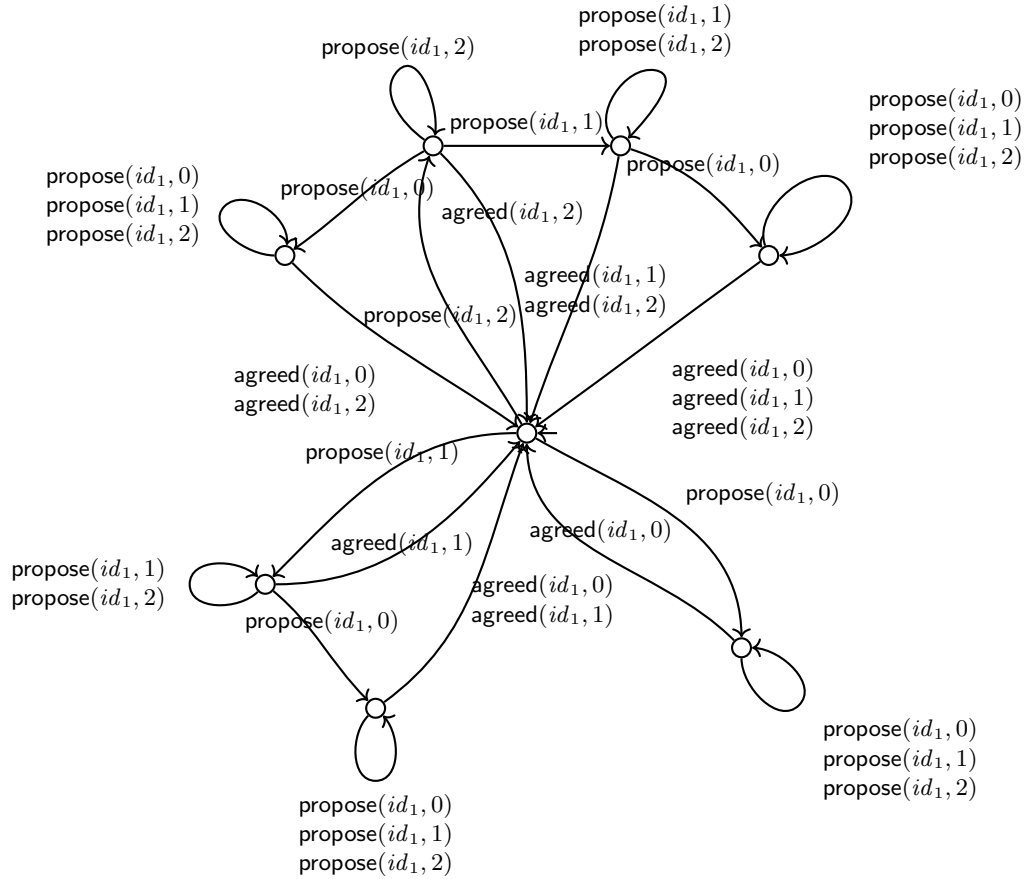
Figure 5: The external behaviour for one single party 1 with three levels of service.

by the following process:

$$\text{proc } SLAN = \text{allow}(\{\text{in}_\text{q}, \text{out}_\text{q}, \text{propose}, \text{agreed}\},$$
$$\text{comm}(\{\text{s}|\text{r}_\text{q} \rightarrow \text{in}_\text{q}, \text{r}|\text{s}_\text{q} \rightarrow \text{out}_\text{q}\},$$
$$P(id_1, id_2, \{\ \}, None, None, false) \ ||$$
$$Q(id_1, [\ ]) \ || \ Q(id_2, [\ ]) \ ||$$
$$P(id_2, id_1, \{\ \}, None, None, false)))$$

4. Behavioural reductions

Given the model it is easy to generate the full state space of the protocol containing its full behaviour, although the sizes of these grow rapidly, see Table 1. After hiding the internal communications, *i.e.*, writing and reading from queues, and, depending on the use case, also the actions by one of the parties, we can apply state space minimisation to better understand our external behaviour and the basic properties of the protocol. The type of behavioural equivalence used when minimising a state space determines which properties are preserved by such a reduction.
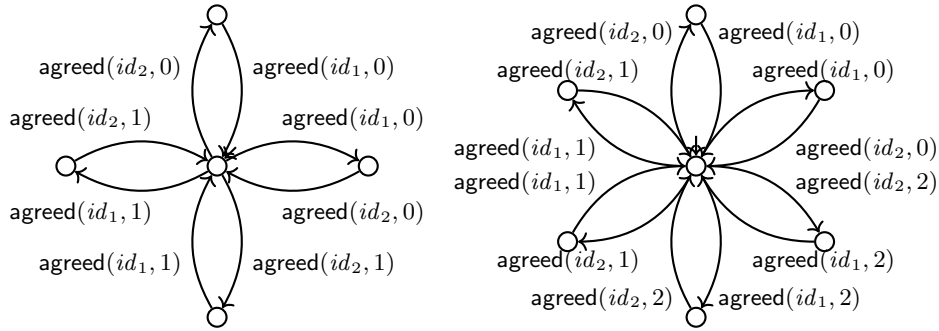
Figure 6: The external behaviour, reduced with respect to weak trace equivalence, for party id_1 with (left) two levels of service, and (right) three levels of service.

Properties (I) and (II) are safety properties on the external traces of the protocol, projected onto the behaviour of a single protocol entity. Safety properties are preserved when minimising with respect to weak trace equivalence. Since the properties are only concerned with **propose** actions of a fixed party, we can abstract from all $\mathsf{in_q}$ and $\mathsf{out_q}$ actions and all **propose** and **agreed** actions by the other party. When we do so, we observe that the minimised state space of our protocol for entity id_1 when $Max = 2$ is exactly as depicted in Figure 2 (see p. 4). The minimised state space of our protocol for entity id_1 when $Max = 3$ is depicted in Figure 5 (see p. 9). In both cases, it is easily seen that properties (I) and (II) hold for these two instances. Obviously, for larger values of Max these pictures become cluttered and therefore rather uninformative.

Property (III) is a safety property that can be visually confirmed by abstracting from all actions of the protocol, except for actions **agreed**. Minimising the state space of the protocol using weak trace equivalence leads to small, insightful pictures. The state space on the left in Figure 6 depicts the resulting state space for $Max = 2$, whereas the state space on the right depicts the state space for $Max = 3$. Both state spaces clearly show that in each round, consisting of two consecutive **agreed** actions, the agreed levels of service are identical.

In case of property (IV), which is a liveness property, a behavioural equivalence relation that is more discriminating than weak trace equivalence is required for minimising the state space to preserve the property. We use divergence-preserving branching bisimilarity [vGLT09] to reduce the state space, abstracting only from actions $\mathsf{in_q}$ and $\mathsf{out_q}$; see Figure 7 for the resulting state space for $Max = 2$. All self-loops have been omitted for the sake of readability. This means that if there is no explicit outgoing transition in a state labelled with a **propose** action, there is a self-loop in that state with such actions. This reduced state space has 25 states and 126 transitions, among which 6 τ-transitions that correspond to reading a message from a queue, inhibiting certain service level agreements, and which are therefore present in the state space even after minimisation.

It may be clear that property (IV) is hard to confirm visually even for such small values of Max. The initial state is the state in the middle. The state at the right corresponds to the system agreeing to a service level 0, and the state at the left corresponds to agreeing to service level 1. The central column (up and down) corresponds to the situation where both protocol entities are in the same round. In the columns at the left and the right, one of the parties still has to report the conclusion of the previous round. In the right upper quadrant party id_1 has still to report agreement on service level 0, in the left upper quadrant it has
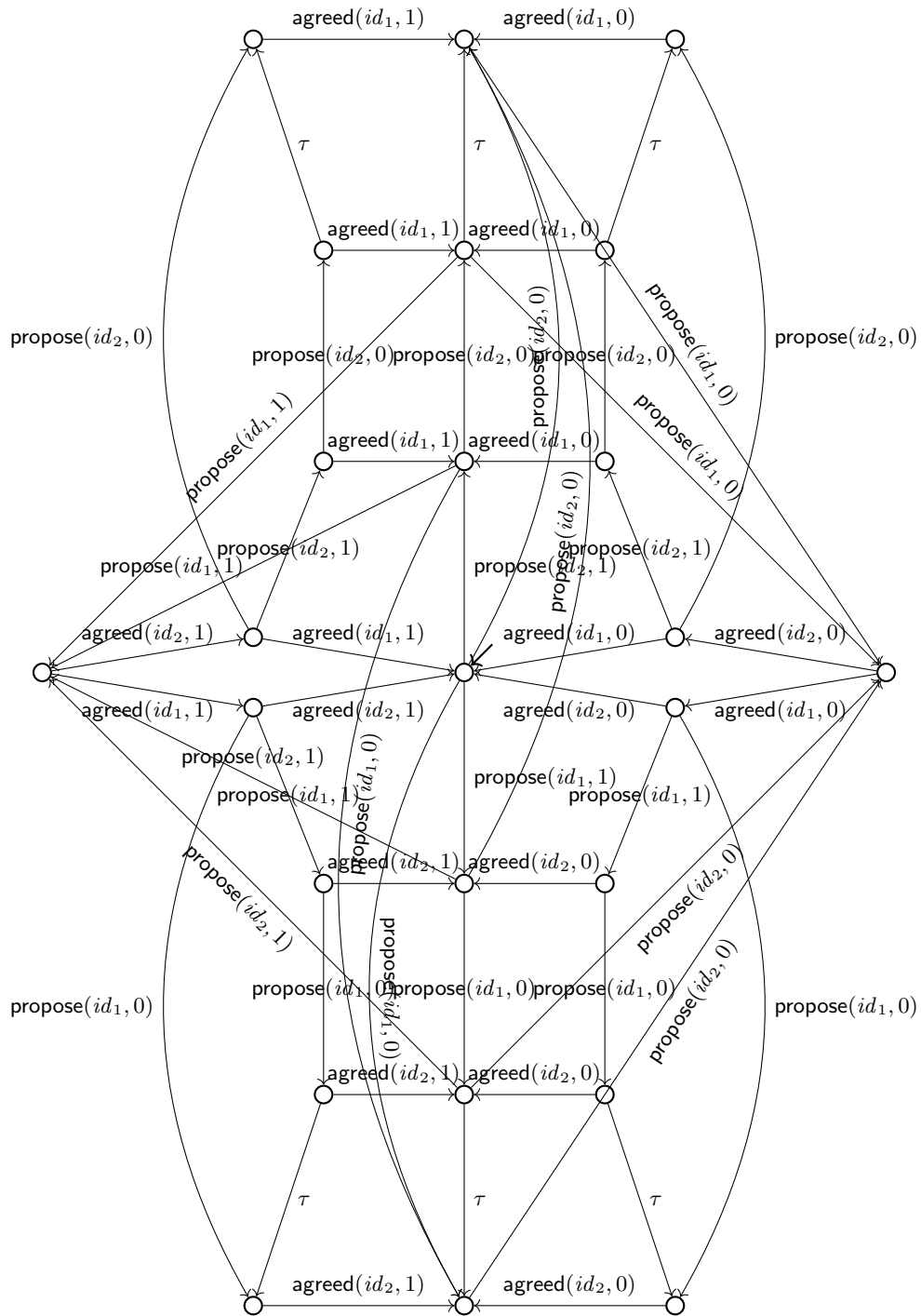
Figure 7: The external behaviour modulo divergence-preserving branching bisimulation for $Max = 2$.

still to report service level 1. In the two lower quadrants protocol entity id_2 still has to report the service level agreed upon.

While moving further up or down, the protocol tends to settle on a lower service level. This is best seen by the transitions going to the rightmost and leftmost states. In the uppermost and lowermost state, transitions to the leftmost state are no longer possible, where the quality of service is settled on level 1. Confirming property (IV) on this transition system is a gruesome task, but feasible with a little perseverance. It is rather undoable to verify requirement (IV) for larger values of Max.

The state space of the protocol also permits to inspect all conceivable contents of the queues. By design, the protocol only submits relevant proposals from a party to the other party, and, therefore, implicitly puts an upper bound on the occupancy of the queues. While our initial assumption was that each queue would contain at most one $decide$ message and Max $inform$ messages, this turns out not to be true. In fact, for the case when $Max = 2$, up to seven simultaneous messages can be present in the queues, as illustrated by the queue contents depicted below.

$$[inform(0), inform(1), decide(1), inform(0), inform(1), decide(1), inform(0)]$$

Note that the leftmost message is the most recent one. The queue corresponds to the situation where one process confirms the decision of two rounds earlier. The rightmost $decide(1)$, marks a decision for two rounds prior. The leftmost $decide(1)$ indicates that also the previous round settled on service level 1; the $inform$ messages in between the two $decide$ messages are those leading to the decision.

The sequence diagram of Figure 8 depicts executions of the protocol that lead to this particular queue contents. We observe that the $inform$ messages in the queue are strictly decreasing in each round, so there can be at most Max consecutive such messages for a single round. The data for three rounds can be simultaneously in the queue. This is nicely illustrated in Figure 8. For the oldest round, agreement has been reached and reported. The oldest $inform$ message, communicating a proposal for the highest service level, has been consumed by id_2. This is needed to reach a decision. For the second round process id_2 sends one $inform$ message, which is sufficient for id_1 to reach a decision. But before reaching a decision, id_1 can already send all, *i.e.*, Max distinct $inform$ messages in the queue, which is then followed by the decision. Although process id_2 did not read any of these messages, and reach a decision in this round, process id_1 can already be fully engaged in the next round, filling the queue again with a whole sequence of Max inform messages.

Summarising, for the oldest round there are $Max - 1$ $inform$ messages, as the receiving party has to have read one $inform$ message to come to an agreement, and one $decide$ message indicating a decision. These are Max messages in total. For the second round there can be Max distinct $inform$ messages and one $decide$ message, totalling $Max + 1$. For the third round there can again be Max distinct $inform$ messages, and no agreement can have been reached, so there is no $decide$ message. If we add up all these messages, we find that there can be at most $3Max + 1$ messages in one queue simultaneously.

5. Formal correctness requirements

In order to convince ourselves that the protocol is correct for larger values of Max, we resort to model checking and formalise the requirements of Section 2 in modal logic. First note that all requirements are essentially requirements on the execution paths of the protocol or,
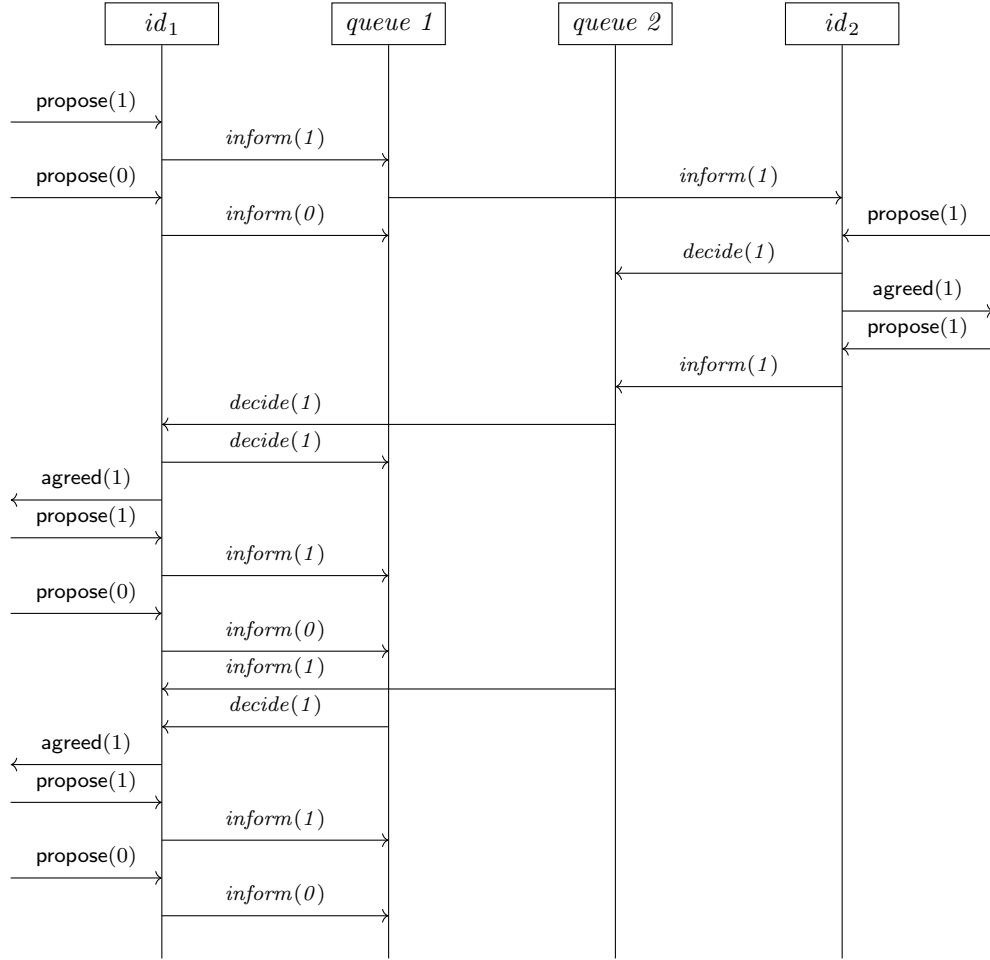
Figure 8: Sequence diagram illustrating that for $Max = 2$, there can be seven messages present in a queue.

in case of requirement (IV), *spines*: execution paths augmented with information concerning enabled actions in states along such paths; moreover, this last property assumes some form of fair behaviour of the two involved parties. Second, note that all properties reason about (sets of) service levels proposed during a round. The former requires a logic capable of reasoning about paths and their branches (*i.e.*, a *branching time logic*); the second benefits from a logic that supports variables for memorising and updating these (sets of) proposals and reasoning about these. Since our model is action-based, we therefore use the modal mu-calculus with data [GM98, GW05a, PWW11] to express the properties.

5.1. **A note on the modal mu-calculus with data.** For a detailed exposition of the syntax and semantics of this logic, including a large number of examples, we refer to [GM14, GKL+19]. We here briefly recall the basic language constructs. *Boolean expressions* and logical operators, such as conjunction, disjunction, recursion variables and first-order quantification, are the basic building blocks of the modal mu-calculus with data. The modal

operators $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ allow for reasoning about the transitions of a transition system: formula $\langle\alpha\rangle\phi$ holds of a state if it it has an α-successor satisfying ϕ, whereas $[\alpha]\phi$ holds of a state if all its α-successors satisfy α. Of particular note here is that α characterises a set of actions: $a(e)$ denotes the singleton set consisting of the parameterised action $a(e)$; the expressions $\alpha\cup\beta$ and $\alpha\cap\beta$ denote set union and set intersection, respectively; for a set described by α, $\bar{\alpha}$ denotes its complement; and $\exists d{:}D.\alpha$ denotes the union of all collections obtained by instantiating d in α. For instance, the formula *true* describes the set of all actions, whereas $\exists n{:}\mathbb{N}.a(n)$ describes the set of actions $\{a(i) \mid i \in \mathbb{N}\}$. The modal mu-calculus derives its expressive power from the fixed point operators μ and ν, which, informally, can be understood as tools to specify finite recursion (*e.g.*, liveness properties) and potentially infinite recursion (*e.g.*, safety properties and invariants). For instance, the absence of deadlock is formalised by $\nu X.([true]X \wedge \langle true\rangle true)$, whereas the formula $\nu X.([true]X \wedge \forall l{:}\mathbb{N}.[\exists id{:}ID.\mathsf{propose}(id,l)](l \in Levels))$ expresses that any process can only propose valid values. In a setting with data, the parameterisation of fixed points enable one to recall, store and update information (data) in recursions.

5.2. Requirement (I).

5.2. **Requirement (I).** The first requirement states that any service level a party agrees to must have been proposed by that party in that round. We model this property by maintaining a finite set *proposed* of service levels that have been proposed by party id in one round. Whenever an action $\mathsf{propose}(id,l)$ takes place the proposed service level l is added to the set *proposed*. Whenever, an action $\mathsf{agreed}(id,l)$ takes place, the agreed level l must be part of this set, and as a new round is started the set *proposed* must be reset. The last line of the modal formula asserts that the requirement holds with *proposed* unmodified when any action, other than $\mathsf{propose}$ or agreed occurs.

$$\forall id{:}ID.\nu X(proposed : Set(\mathbb{N}) = \emptyset).$$
$$(\forall l{:}\mathbb{N}.[\mathsf{propose}(id,l)]X(proposed \cup \{l\})) \wedge$$
$$(\forall l{:}\mathbb{N}.[\mathsf{agreed}(id,l)](l \in proposed \wedge X(\emptyset))) \wedge$$
$$([\exists l{:}\mathbb{N}.\mathsf{propose}(id,l)\cup\mathsf{agreed}(id,l)]X(proposed)).$$

We remark that the parameterisation of fixed points is not strictly needed; *i.e.*, the property can also be formalised by a more 'classical' formula with nested fixed points. But we find that using data in modal mu-calculus formulas makes the formulas easier to understand as data parameters often match natural concepts.

5.3. Requirement (II).

5.3. **Requirement (II).** The second requirement expresses that any party id proposing a service level that is higher than the lowest service level already proposed by the party cannot supersede that lower service level. The service level that is agreed upon is therefore never of a level that is rejected in the negotiation process. This formula is similar in spirit to the previous formula, except that, in a round, proposed service levels are only added to the set *proposed* if they are lower than the current minimum, and an additional set *rejected* is maintained to keep track of all the rejected levels that have been proposed. Note that this set does not simply consist of the set of levels that are larger than the least level proposed so far, since a service level higher than this least level may still be under consideration if it has been proposed at an earlier stage. We can therefore only reject proposals for service levels that have not previously been proposed and that are not lower than the current minimum;

these are exactly those service levels added to a set *rejected*. The formula then asserts that no service level of the set *rejected* can be agreed upon in a round.

$\forall id{:}ID.\nu X(proposed : Set(\mathbb{N}) = \emptyset, rejected : Set(\mathbb{N}) = \emptyset).$
$\quad (\forall l{:}\mathbb{N}.[\mathsf{propose}(id,l)](\,(l\notin proposed \wedge l{>}min(proposed) \wedge X(proposed, rejected\cup\{l\})) \vee$
$\qquad\qquad\qquad (l\in proposed \wedge X(proposed, rejected)) \vee$
$\qquad\qquad\qquad (l{\leq}min(proposed) \wedge X(proposed\cup\{l\}, rejected))\,)\,) \wedge$
$\quad (\forall l{:}\mathbb{N}.[\mathsf{agreed}(id,l)](l\notin rejected \wedge X(\emptyset,\emptyset))\,) \wedge$
$\quad ([\exists l{:}\mathbb{N}.\mathsf{propose}(id,l)\cup\mathsf{agreed}(id,l)]X(proposed, rejected)).$

5.4. Requirement (III).

Requirement (III) states that if a service level l is agreed upon during a round, then both parties agree on the same service level. As the rounds are asynchronous, it can be the case that in the current round, no service level has been agreed upon yet (*i.e.*, no $\mathsf{agreed}(id,q)$ action for the current round has happened), or one of the parties has already reported on a service level that is agreed upon (*i.e.*, some $\mathsf{agreed}(id,l)$ has been reported in the current round). To account for this asynchronicity, we therefore use parameters a_1 and a_2 of type $\mathbb{N}$ to store the service level agreed upon in a round by party id_1, resp. id_2, and that contain *None* if no such value for the current round has been announced yet. Whenever $\mathsf{agreed}(id,l)$ is reported by party id, the announced value must match that of the other party if it has been announced (*i.e.*, if the associated parameter is different from *None*).

$\nu X(a_1 : \mathbb{N} = None, a_2 : \mathbb{N} = None).$
$\quad (\forall l{:}\mathbb{N}.[\mathsf{agreed}(id_1,l)]((a_2\not\approx None \wedge l\approx a_2 \wedge X(None, None)) \vee (a_1\approx a_2\approx None \wedge X(l, a_2)))) \wedge$
$\quad (\forall l{:}\mathbb{N}.[\mathsf{agreed}(id_2,l)]((a_1\not\approx None \wedge l\approx a_1 \wedge X(None, None)) \vee (a_1\approx a_2\approx None \wedge X(a_1, l)))) \wedge$
$\quad ([\exists id{:}ID, l{:}\mathbb{N}.\mathsf{agreed}(id,l)]X(a_1, a_2)).$

5.5. Requirement (IV).

The fourth requirement states that whenever matching proposals have been made, inevitably both parties will be informed about an agreed quality of service. Since the protocol always returns to a state in which it is able to accept new proposals, the agreement can be postponed indefinitely by proposing service levels that will nevertheless be ignored. Once the protocol is in a state in which an agreement can be sent, however, it remains in such a state until it has actually been sent.

We note that both parties may potentially agree to a service level that is below their currently matching proposals. This is the case if one party submits a lower level after earlier matching proposals have been submitted, or if one of the parties also submitted proposals for lower service levels. For instance, if party id_1 first proposes service level 2, followed by 1, and party id_2 then responds by submitting service level 2, it may still be the case that party id_1 lowers the desired service level even further by submitting 0. In such a case agreement might then only be possible if id_2 submits level 0 as well, even though level 0 was not among the service levels at the time both parties first submitted a matching proposal.

We formulate this property using two sets of proposed quality of services and whenever the intersection of these sets is not empty, it is required that agreed actions must take place.

However, due to asynchronicity, both protocol entities may be in different rounds. Therefore, we keep lists of finite sets of proposed quality of services q_1 for process 1 and q_2 for process 2. The first element in the list contains the set of quality of services proposed in the current round that is not yet finished by both parties. It can be that one of the processes

is already collecting proposals for the next round. In that case its list contains two elements and newly proposed quality of services are inserted in the second set. Note that the sum of the lengths of the lists of q_1 and q_2 can be 2 or 3.

In order to understand the formula after the condition $head(q_1) \not\approx None \wedge head(q_2) \not\approx None$, it is useful to understand the following simpler formula:

$$\mu Z.\nu Y.([\overline{a \cup b}]Z \wedge [b]Y \wedge \langle \overline{b} \rangle true).$$

This formula says that action a will become enabled within a finite number of non-b steps but any b-action may indefinitely postpone this moment; until this moment, non-b steps remain enabled and following these brings the inevitability of a closer.

In our particular setting, the set of actions represented by a and b is not static but dynamic; the reason is that before agreement becomes persistently enabled, the parties may still be negotiating and the final agreement is therefore not yet settled in all circumstances. Our formula therefore asserts that agreement becomes inevitable—if it is not decided already—when one party proposes to lower the service level to one that matches the other party's minimum level. We keep track of these minimum proposed levels by means of two parameters of the Z and Y fixed point variables, and update these parameters with every occurrence of a propose action.

In the formulation below, the action a takes the role of the agreed(id, q) action (for universally quantified id). The set of actions represented by b within the must-modality is replaced by the set of all propose actions, and the set of b actions within the may-modality is replaced by all propose actions, except for the possibly necessary proposal matching the minimum of the proposals of the other party. The outermost fixed point X is used to assert that the inevitability of agreements is an invariant. Both parameters q_1 and q_2 are necessary to keep track of the current minimal proposals made by the two parties in a round. Due to asynchronicity, we may need information about the minimal level proposed in a previous round. To this end, we use lists of service levels.

$$(\nu X(q_1{:}List(\mathbb{N}) = [None], q_2{:}List(\mathbb{N}) = [None]).$$
$$(\forall l{:}\mathbb{N}.[\mathsf{propose}(id_1, l)]X(rtail(q_1) \triangleleft min(rhead(q_1), l), q_2)) \wedge$$
$$(\forall l{:}\mathbb{N}.[\mathsf{propose}(id_2, l)]X(q_1, rtail(q_2) \triangleleft min(rhead(q_2), l))) \wedge$$
$$([\overline{\exists l{:}\mathbb{N}.\mathsf{agreed}(id_1, l)}](((|q_2| \not\approx 1 \wedge X(tail(q_1) \triangleleft None, tail(q_2))) \vee (|q_2| \approx 1 \wedge X(q_1 \triangleleft None, q_2)))) \wedge$$
$$([\overline{\exists l{:}\mathbb{N}.\mathsf{agreed}(id_2, l)}](((|q_1| \not\approx 1 \wedge X(tail(q_1), tail(q_2) \triangleleft None)) \vee (|q_1| \approx 1 \wedge X(q_1, q_2 \triangleleft None)))) \wedge$$
$$([\overline{\exists id{:}ID, l{:}\mathbb{N}.\mathsf{propose}(id, l) \cup \mathsf{agreed}(id, l)}]X(q_1, q_2)) \wedge$$
$$((head(q_1) \not\approx None \wedge head(q_2) \not\approx None) \Rightarrow$$
$$\quad \forall id{:}ID.((id \approx id_1 \wedge |q_1| \approx 1) \vee (id \approx id_2 \wedge |q_2| \approx 1)) \Rightarrow$$
$$\quad\quad \mu Z(m_1'{:}\mathbb{N} = head(q_1), m_2'{:}\mathbb{N} = head(q_2)).$$
$$\quad\quad\quad \nu Y(m_1{:}\mathbb{N} = m_1', m_2{:}\mathbb{N} = m_2').$$
$$\quad\quad\quad\quad ([\overline{\exists l{:}\mathbb{N}.\mathsf{agreed}(id, l) \cup \exists id'{:}ID.\mathsf{propose}(id', l)}]Z(m_1, m_2) \wedge$$
$$\quad\quad\quad\quad (\forall l{:}\mathbb{N}.[\mathsf{propose}(id_1, l)]Y(min(m_1, l), m_2)) \wedge$$
$$\quad\quad\quad\quad (\forall l{:}\mathbb{N}.[\mathsf{propose}(id_2, l)]Y(m_1, min(m_2, l))) \wedge$$
$$\quad\quad\quad\quad (\langle \overline{\exists l{:}\mathbb{N}.((l \not\approx m_2 \cup m_1 \leq m_2) \cap \mathsf{propose}(id_1, l))} \cap$$
$$\quad\quad\quad\quad\quad \overline{\exists l{:}\mathbb{N}.((l \not\approx m_1 \cup m_2 \leq m_1) \cap \mathsf{propose}(id_2, l))} \rangle true)))$$

5.6. **Verification.** The four properties above have been verified for *Max* ranging from 1 up-to and including 5 using the mCRL2 toolset [BGK$^+$19]. All four properties hold for all these five instances. We note that also the two properties stated in Section 5.1 hold for these five instances. There are several ways of performing such a verification. Due to the relatively elegant external behaviour of the protocol, it turns out that it is most fruitful to first generate the state space for each instance, reduce these modulo divergence-preserving branching bisimulation [vGLT09], which preserves the validity of the formulas, and verify the modal formulas on the reduced state spaces. In Table 1 we indicate the sizes of the generated state space. Note that these grow quickly with increasing *Max*. Suffices 'k', 'M' and 'G' stand for 10^3, 10^6 and 10^9, respectively. The column labelled 'mod dpbb' depicts the sizes after reduction modulo divergence-preserving branching bisimulation. Note the very substantial reductions.

Max	#states	#transitions	#states mod dpbb	#transitions mod dpbb
1	233	746	8	22
2	7.9k	38k	25	126
3	239k	1.6M	66	482
4	6.7M	57M	163	1550
5	176M	1.8G	388	4.5k
6	4.4G	541G	901	12.3k

Table 1: Sizes of the state spaces for various values of *Max*

6. Conclusions and future work

We designed a symmetric protocol for repeatedly negotiating a desired quality of service level between two parties. We spent considerable effort in ensuring that the state space of the external behaviour of our protocol is as 'small' and 'insightful' as possible. This is to say that we did not manage to obtain smaller state spaces without violating the listed desirable properties or by inadvertently introducing deadlocks. It is, however, reasonably straightforward to make changes that yield more complex state spaces (also after reduction) and that even satisfy all four modal formulas expressing the correctness of our protocol. We found that due to the concise external behaviour, we could verify the modal formulas for very substantial state spaces as these state spaces reduce very considerably. This is in line with one of the design rules for verifiable systems as mentioned in [GKO15].

There are a few directions for future work. For instance, the protocol can be generalised to serve n parties. Furthermore, it is an open question how to prove the validity of our modal formulas for our protocol for arbitrary value for *Max*. The complexity of the protocol is such that any convincing correctness argument requires mechanical proof checking or full automation. One option can be to develop proof methods for parameterised Boolean equation systems [GW05b], inspired by the *cones and foci* method [GS01, FP03], which is very effective for proving an implementation behaviourally equivalent to a specification for n processes and infinite data domains, and which can be mechanised in a proof checker [FPvdP06]. Other options, which, in theory, work for properties (I)–(III), can be to automate proof methods based on invariants [OW10] or techniques such as abstract interpretation [CGWW15]. However, for property (IV), it is conceivable that any such technique will fail due to the

presence of the least fixed point operator, see *e.g.* [CGWW15], thus calling for even stronger proof methods.

Acknowledgements

References

[BdM06]   Cyril Briquet and Pierre-Arnoul de Marneffe. Grid resource negotiation: Survey with a machine learning perspective. In *Parallel and Distributed Computing and Networks*, pages 17–22. IASTED/ACTA Press, 2006.

[BGK⁺19]   Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *TACAS (2)*, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019.

[CGWW15]   Sjoerd Cranen, Maciej Gazda, Wieger Wesselink, and Tim A. C. Willemse. Abstraction in fixpoint logic. *ACM Trans. Comput. Log.*, 16(4):29:1–29:39, 2015.

[CKML13]   Mohamed Aymen Chalouf, Francine Krief, Nader Mbarek, and Tayeb Lemlouma. Improvement of a service level negotiation protocol using formal verification. In *ISCC*, pages 783–788. IEEE Computer Society, 2013.

[CLW15]   Sjoerd Cranen, Bas Luttik, and Tim A. C. Willemse. Evidence for fixpoint logic. In *CSL*, volume 41 of *LIPIcs*, pages 78–93. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[DDO06]   George Dimitoglou, Okan Duzyol, and Lawrence Owusu. Formal verification of a simple automated negotiation protocol. In *Software Engineering Research and Practice*, pages 650–656. CSREA Press, 2006.

[FLP85]   Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FP03]   Wan Fokkink and Jun Pang. Cones and foci for protocol verification revisited. In Andrew D. Gordon, editor, *FOSSACS*, volume 2620 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 2003.

[FPvdP06]   Wan J. Fokkink, Jun Pang, and Jaco van de Pol. Cones and foci: A mechanical framework for protocol verification. *Formal Methods Syst. Des.*, 29(1):1–31, 2006.

[GKL⁺19]   Jan Friso Groote, Jeroen J.A. Keiren, Bas Luttik, Erik P. de Vink, and Tim A.C. Willemse. Modelling and analysing software in mCRL2. In *FACS*, volume 12018 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2019.

[GKO15]   Jan Friso Groote, Tim W. D. M. Kouters, and Ammar Osaiweran. Specification guidelines to avoid the state space explosion problem. *Softw. Test., Verif. Reliab.*, 25(1):4–33, 2015.

[GM98]   Jan Friso Groote and Radu Mateescu. Verification of temporal properties of processes in a setting with data. In *AMAST*, volume 1548 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1998.

[GM14]   Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.

[GS01]   Jan Friso Groote and Jan Springintveld. Focus points and convergent process operators: a proof strategy for protocol verification. *J. Log. Algebr. Program.*, 49(1-2):31–60, 2001.

[GW05a]   Jan Friso Groote and Tim A. C. Willemse. Model-checking processes with data. *Sci. Comput. Program.*, 56(3):251–273, 2005.

[GW05b]   Jan Friso Groote and Tim A. C. Willemse. Parameterised Boolean equation systems. *Theor. Comput. Sci.*, 343(3):332–369, 2005.

[NRG19]   Thomas Neele, Marijn Rol, and Jan Friso Groote. Verifying system-wide properties of industrial component-based software. In Hossein Hojjat and Mieke Massink, editors, *FSEN*, volume 11761 of *Lecture Notes in Computer Science*, pages 158–175. Springer, 2019.

[OTM+07]   Michel A. Oey, Reinier J. Timmer, David G. A. Mobach, Benno J. Overeinder, and Frances M. T. Brazier. Ws-agreement based resource negotiation in agentscape. In *AAMAS*, page 267. IFAAMAS, 2007.

[OW10]   Simona Orzan and Tim A. C. Willemse. Invariants for parameterised Boolean equation systems. *Theor. Comput. Sci.*, 411(11-13):1338–1371, 2010.

[PKB06]   Michael S. Parkin, Dean Kuo, and John M. Brooke. A framework & negotiation protocol for service contracts. In *IEEE SCC*, pages 253–256. IEEE Computer Society, 2006.

[PWW11]   Bas Ploeger, Wieger Wesselink, and Tim A. C. Willemse. Verification of reactive systems via instantiation of parameterised Boolean equation systems. *Inf. Comput.*, 209(4):637–663, 2011.

[vGLT09]   Rob J. van Glabbeek, Bas Luttik, and Nikola Trcka. Branching bisimilarity with explicit divergence. *Fundam. Inform.*, 93(4):371–392, 2009.

[WW18]   Wieger Wesselink and Tim A. C. Willemse. Evidence extraction from parameterised Boolean equation systems. In Christoph Benzmüller and Jens Otten, editors, *ARQNL*, volume 2095 of *CEUR Workshop Proceedings*, pages 86–100. CEUR-WS.org, 2018.

[YYWL12]   Edwin Yaqub, Ramin Yahyapour, Philipp Wieder, and Kuan Lu. A protocol development framework for SLA negotiations in cloud and service computing. In *GECON*, volume 7714 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2012.