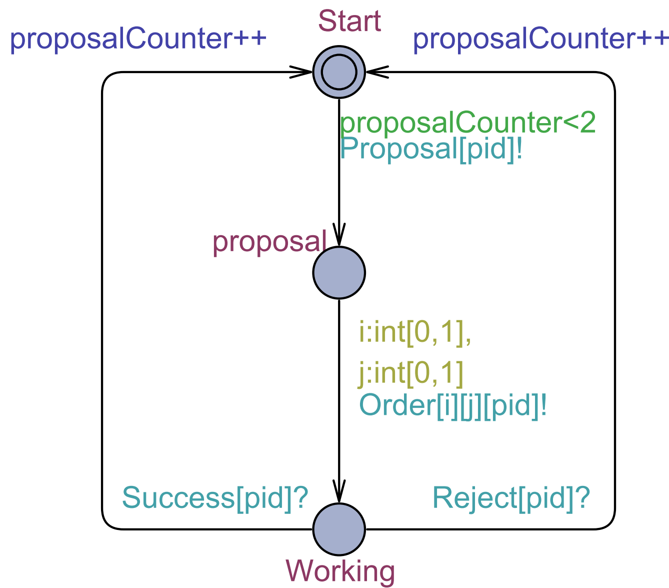


Fig. 2. The customer process.

### B. Behaviour of a Customer Process

In Fig. 2, the automaton for *Customer* process is illustrated. The initial state is named as *Start*. The *Customer* process has three states. The first state is *Start* state, second is *Proposal* state and the third state is *Working* state. There are four major actions in this process described below:

1) Sending proposal to the *Logistics*.
2) Sending Order to the *Logistics* and waiting for the response.
3) Going to the start state through rejected path if response is negative from *Logistics*.
4) Going to the start state through success path if order is successfully completed.

First of all, the channel *Proposal* transfers a value to some logistic agent which is originated by a customer process. The logistic agents receives these values while synchronization of channel *Proposal[x]*. Here ‘x’ represents process ID (pid), i.e., the customer ID sending a proposal.

Secondly, after sending proposal the customer sends order using channel *Order[2][2]* which is received by the logistics at channel *Order[i][j][cus_id]* and goes to the *Working* state. There are two values ’i’ and ’j’ that are sent by the *Customer* for the activities that a customer needs. If *i=0* and *j=1* the *Customer* needs activity *j*, if *i=1* and *j=0* the customer needs activity *i* and if both *i* and *j* are 1 the customer needs both the activities.

At the end if the customer receives error message from the logistics that the order cannot be processed or teams fail to work then it goes to *Start* state using *Reject* channel, if the work is successfully done it goes to *Start* state using *Success* channel. On initial state means that it is ready for the next proposal. There is a counter *proposalCounter* for the proposals sent by a *Customer*.

### C. Automaton for the Logistics Process

In Fig. 3 the automaton for *Logistics* process is illustrated. The initial state is named as *Start*. There are five major actions in this process described below:

1) Receiving proposal from the *Customer* and decompose it.
2) Forming small team of contractors that will execute the activities.
3) Providing alternative if small team has issue in the order.
4) Providing alternative contractor if one team needs alternative and other one ready to work.
5) Providing alternative contractor if one team fails to complete its work and other one successfully completed work.

The Fig. 4 shows that the *Logistics* process receives proposal using synchronization channel *Proposal[x]*. These values are process ID of *Customer* describes that which customer sends order. After receiving proposal the *Logistics* receives order from the *Customer* using channel *Order[i][j][cus_id]?*. There are two values ’i’ and ’j’ that are received by the *Logistics* are the activities that customer needs. If *i=0* and *j=1* the *Customer* needs activity *j*, if *i=1* and *j=0* the *Customer* needs activity *i* and if both *i* and *j* are 1 the *Customer* needs both the activities.

After receiving order a *Logistic agent* tries to rank contractors according to the activities a *Customer* demands. For example if customer needs A1 activity then contractor that can perform A1 activity is not available then the order is rejected and *Logistics* goes to *Start* state, ready to receive new order and acknowledges the *Customer*. Similarly, if *Customer* needs A1 and A2 activities contractors for both activities should be available. If contractors successfully ranked *Logistics* assign activities to contractors and goes to *ContractorRanked* State.

Fig. 5 shows the next part of the *Logistics* process. After ranking the contractors *Logistics* waits for the response from the small team whether or not they will accept the contract. This is done by sending each activity to that small team which is available and willing to do work. For this purpose *NegT1[pid][0]!* and *NegT2[pid][1]!* channels are used for *SmallTeam(0)* and *SmallTeam(1)*, respectively where *[pid]* is the process id of *Logistics* sending order and *[0]* and *[1]* values describe the pid of *SmallTeam* to which *Logistics* are sending order. The response from the *SmallTeam* can be of three types *Logistics* receives it on *SmallTeam* state which is as follows:

1) Both the *Small Teams* are ready to do work or committed.
2) *Small Team 1* needs alternative and *Small Team 2* ready to do work.
3) *Small Team 1* ready to do work and *Small Team 2* needs alternative.

If both the *Small Teams* are ready to do work. Then the *Logistics* receive the response using channels *Commited[pid][c]?* from both teams. *Commitcount++* is used to count the commit response, if the value in *Commitcount* is 2 it means both teams committed in case of *Customer* needs one activity the value of *Commitcount* will be 1. If *Small Team 1* needs alternative

Fig. 3. The logistics process.

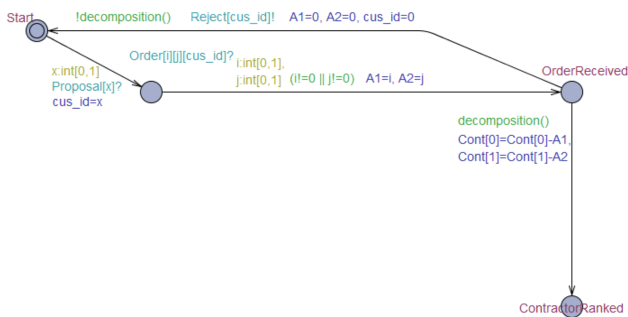


Fig. 4. The logistics process (a).

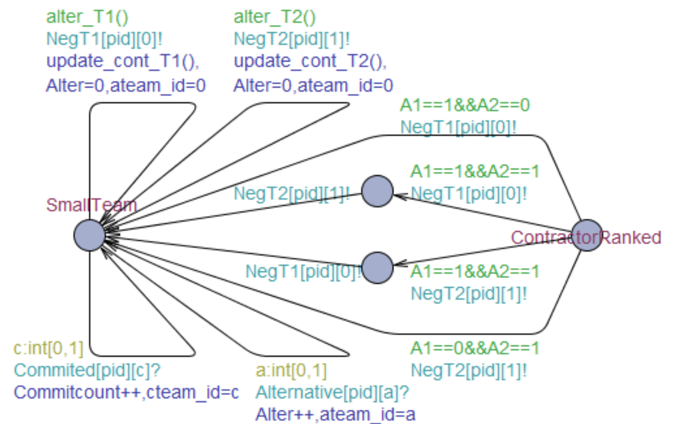


Fig. 5. The logistics process (b).

and *Small Team 2* ready to do work or vice versa then *Commited[pid][c]?* for committed and *Alternative[pid][a]?* for alternative response is used. *Commitcount++* used to count the commit response and *Alter++* used to count the alternative response.

When one team is committed and other needs alternative then *Logistics* checks for the available contractors willing to work and assign activity for which *Small Team* needs an alternative. This is done by using *NegT1[pid][0]!* and *NegT2[pid][1]!* for *Small Team 1* and *Small Team 2*, respectively.

Fig. 6 shows the next procedure after small team formation. If both the teams need alternative in case *Customer* needs both activity then process goes to *AlternativeNeeded* state. Moreover, it goes to *Start* state for negative acknowledgment to customer using channel *Reject[cus_id]!* after which it becomes ready for receiving new order. In case of small Teams are committed to work then process goes to *Contractorcommitted* state. At this state *Logistics* checks whether small teams have complete their work or not. If the teams complete their work successfully then the respective logistic agent goes to start state using channel *Success[cus_id]* and acknowledges the *Customer* accordingly. The number of failed teams are counted

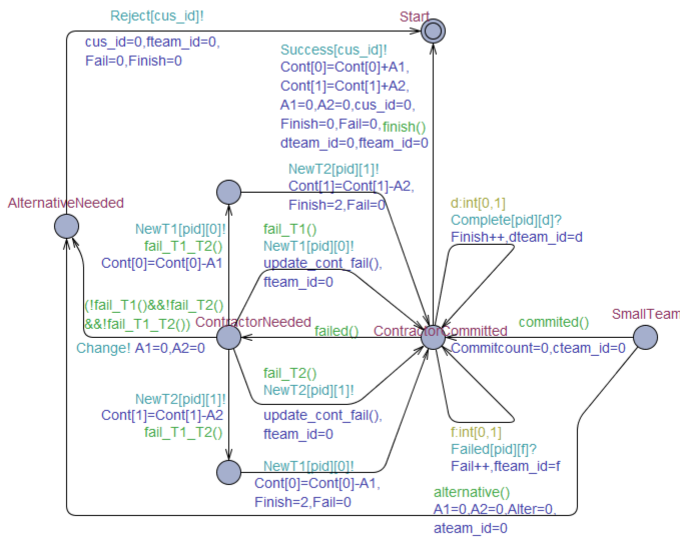


Fig. 6. The logistics process (c).

by *Fail*.

When one team is successful and other team fails then *Logistics* checks for the available contractors that are willing to do work and assign activity for which *Small Team* fails. This is done by using *NewT1[pid][0]!* and *NewT2[pid][1]!* for *Small Team 1* and *Small Team 2*, respectively. If the contractor is available activity assigns to that contractor and after completing work *Logistics* process goes to *ContractorCommitted* state and further more at *Start* state. In case of contractors are not available then process goes to *AlternativeNeeded* using channel *Change!* this transition further more goes to start state using channel *Rejected[cus_id]!*.

When we use UPPAAL system models, functions can be declared within the procedure or process. We can pass parameters in functions and functions can also have return type. The *Logistics Process* have various functions and are used at different transitions to perform its functionality.

1. *Decomposition():* function used as guard and checks the contractor against activities. *A1* and *A2* are the activities. If customer needs both activities vales of *A1* and *A2* will be 1, if customer needs one activity then the value of *A1* and *A2* will be 1 according to the activity that customer needs. This guard prevents to take action if the contractors will not available against the activity which customer needs and take action that goes to start state so logistics can receive new order.

2. *Committed():* function is also used as guard and checks the small teams response. If customer needs both the activities then both the teams should be committed to work if not, guard will prevent to goes to *ContractorCommitted* state. If customer needs one activity then the small team against that activity should be committed. This Function uses an integer variable *CommitCount* for counting the response form the the teams and compares with number of teams and activities.

3. *Alternative():* function is also used to check the small team response. It works same as *committed()* function but the difference is that in case of both activities, if both the teams need alternative this guard will allow to go to *Alternative-Needed* state through alternative transition. And if customer needs one activity then the small team against that activity can ask for alternate. This Function uses an integer variable *Alter* for counting the response form the the teams and compares with number of teams and activities.

4. *Alter_T2():* function is used to check the availability of contractors. If the customer needs both the activities and 1st team committed and 2nd team needs alternative, then contractor for 2nd team should be available otherwise this guard will prevents to take further action and wait for the availability. Statement *Cont[1]¿0* checks the availability. This function uses *Commitcount* and *Alter* variables to check which teams needs alternative.

5. *Alter_T1():* function is also used to check the availability of contractors. If the customer needs both the activities and 1st team needs alternative and 2nd team is committed, then contractor for 1st team should be available otherwise this guard will prevent to take further actions and waits for the availability. Statement *Cont[0]¿0* checks the availability. This function uses *Commitcount* and *Alter* variables to check which teams needs alternative.

6. *Finish():* function is used when small team complete their work successfully after commitment. In case of customer needs one activity, variable *Finish* value will be 1 and function allows to finish the work and *Logistics* goes to *Start* state to take new order. If customer needs both activities value of *Finish* will be 2.

7. *Failed():* function is used if the small team fails to complete work after commitment. Both the teams can be failed or may be one team fail and other complete the task then this function will allows to take action and goes to *ContractorNeeded* state. *Fail* and *Finish* variables are used to check which teams are failed or has completed their work. If value of *Fail* is 2 then both the teams failed, if value of *Fail* and *Finish* is 1 then one team has completed and other has finished the work.

8. *Fail_T2():* function is used to check the availability of contractors in case of one team finishes its work and other team completes its work successfully. If the customer needs both the activities and 1st team finished work and 2nd team failed, then contractor for 2nd team should be available for replacement otherwise this guard will prevents and goes to *AlternativeNeeded* state. In case if a customer needs only 2nd activity and small team fails to complete work then contractor against that activity should be available for replacement. Statement *Cont[1]¿0* checks the availability. This function uses *fteam_id* variable to check which teams is failed.

9. *Fail_T1():* function works same as *fail_T2()* difference is that if the customer needs both the activities and 1st team failed and 2nd team finished, then contractor for 1st team should be available for replacement otherwise this guard will prevents and goes to *AlternativeNeeded* state. And if customer needs only 1st activity and small team fails to complete work then contractor against that activity should be available for replacement. Statement *Cont[0]¿0* checks the availability.

10. *Fail_T1_T2():* function is used if customer needs both activities and both are failed to complete their work after

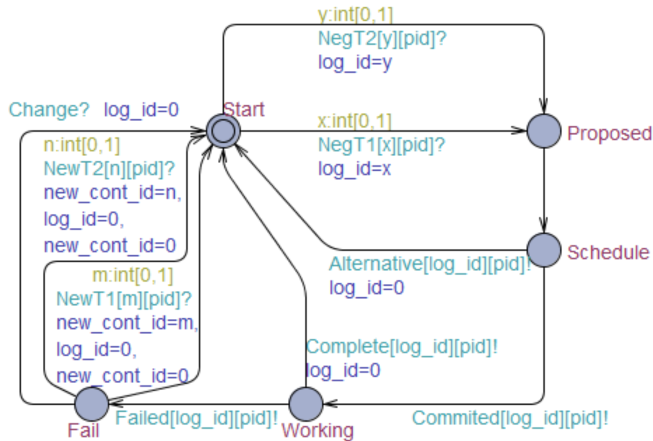


Fig. 7. The small team process.

commitment then both contractors for 1st team and 2nd team should be available for replacement otherwise this guard will prevents and goes to *AlternativeNeeded* state. Statement *Cont[0]¿0* and *Cont[1]¿0* checks the availability.

11. *Update_cont_T1()* and *update_cont_T2():* functions are used to update contractors. If customer needs both activities and one team is committed and other needs alternative. The function used *cteam_id*, *ateam_id* and *Alter* variables to check which team is committed and which one needs alternative.

12. *Update_cont_fail:* function is used to update contractors in case of customer needs both activities and one team has completed its work successfully and other team failed to work. This function uses same variables as previous functions.

### D. The Automaton for the Small Team Process

Fig. 7 the automaton for small team process is illustrated. The initial state is named as *Start*. The *Logistics* process has five states. The first state is *Start* state, second is *Proposed*, third is *Schedule*, fourth is *Working* and the fifth state is *Fail* state. There are four major actions in this process described below:

1) Receiving proposal from the logistics.
2) Sending acknowledgment to the logistics if needs some changing or alternative proposal.
3) Sending acknowledgment to the logistics if ready to perform activity and goes to *Working* state.
4) If fails to complete activity goes to fail state and waiting for new contractor who is willing to complete that activity.

The channel *NegT1[x][pid]?* and *NegT2[y][pid]?* receives value from the logistics process, received values are the tasks assigned to the team 1 and team 2 received from channels *NegT1[x][pid]?* and *NegT2[y][pid]?*, respectively. The small team checks its schedule if team has no issue and willing to work. Then this small team acknowledges the logistics using *Commited[log_id][pid]!* channel to go to *Working* state. If there are issues in the proposal like small team has not enough time or could not perform that activity on time, small team acknowledges the logistics that it needs alternative for that task using *Alternative[log_id][pid]!* channel and goes to *Start* state for receiving new or alternative proposal.

After committing small team starts working on the task. If small team fails to complete its task it sends negative acknowledgment to the logistics that it needs new contractor and goes to *Fail* state waiting for new contractor to be assigned by the logistics and this is done by using *Failed[log_id][pid]!* channel. If the contractor is available and willing to work small team is assigned that contractor using *NewT1[m][pid]?* and *NewT2[n][pid]?* channels for team 1 and team 2, respectively otherwise goes to *Start* state after receiving response from the logistics using *Change?* channel.

## IV. Functional Requirements

According to [3], we extract the following functional requirements:

R1: Deadlock freedom. No deadlock when a customer needs to places an order. In other words, deadlock can occur only when there are no more orders.

R2: If customer sends order, logistic agents eventually acknowledge it.

R3: A customer is in working state after paying an order.

R4: If logistics agent is in OrderReceived state if it receives an order.

R5: Every order decomposed by some logistic agent results in formulating a small team.

### A. Formal Specification of Requirements

In this section, we describe formal specification of the requirements. The customer process sends order and then increases its counter, i.e., known as *proposalCounter*. This increment continues up to two it means the customer can send maximum 2 orders. So, according to the R1 requirement, there is deadlock only when there are no more orders to send by the customers. The formula of R1 requirement is given below.

```
A[] deadlock imply(Customer(0).
    proposalCounter==2&&Customer(1).
    proposalCounter==2))
```

When customer sends order the logistics agent receives and acknowledges it with a message either the given order is workable or not. The formula of to represent this requirement is:

```
E<> forall (i:id_t) forall (j:id_t)
(Customer(i).Working &&Logistics(j).OrderReceived)
```

Formula describes that Customer(0) and Customer(1) sends proposal to Logistics(0) and Logistics(1) and vice versa. The logistics acknowledges the customer.

When customer sends order the logistic agents receives and acknowledges the customer at that time customer goes to *Working* state. For example, when customer(0) sending order definitely goes to *working* state. The formula of this requirement is.

```
forall (i:id_t)Customer(i).proposal -->
          Customer(i).Working
```

According to the R4 requirement, when a logistics agent receives proposal it goes to *OrderReceived* state. The formula of the requirement is given below.

```
forall (i:id_t) Logistics(i).proposalReceived
                --> Logistics(i).OrderReceived
```

According to the R5 requirement, every order decomposed by some logistics agent formulates a small team. The formula of this requirement is given below.

```
forall (i:id_t) Logistics(i).ContractorRanked
                --> Logistics(i).SmallTeam
```

## V. Results

To analyse features specified in the above section, we use the verifier, a feature of UPPAAL model checker. Ultimate results are derived in query section of verifier feature and presented in Table I. In query section, the feature is written and its consequences are to be revealed in the status section. The outcomes are in the form of "Satisfied" and "Not Satisfied" of property. We verify our system model for,

**Total Number of Processes = 3**

**Order Sending Limit = 2**

**Activity Demand Limit = 2**

TABLE I. Results

| Requirement | Status | Computational Time |
|---|---|---|
| **R1** | Not Satisfied, 131 states | 0.125 sec |
| **R2** | Satisfied, 28,180 KB | 0.015 sec |
| **R3** | Satisfied, 138 states | 0.539 sec |
| **R4** | Satisfied, 1623 states | 0.562 sec |
| **R5** | Not Satisfied, 32,204 KB | 0.032 sec |

R1: This requirement is violated and not satisfied. According to the requirement system should be deadlock free or deadlock can occur only when there are no more orders to send. But there is a scenario in which this requirement is not satisfied. When a small team needs alternative there is no more contractor available against that activity at that state deadlock occurs. The counter example for requirement R1 generated by UPPAAL is shown in Fig. 8.

R5 requirement is not satisfied and according to this requirement upon decomposing an order by logistic agents, small team is formed. If a customer needs both activities then upon decomposition if one small team needs alternative but there contractors are unavailable pertaining to that activity then small team is not formed, so this requirements is not satisfied. The counter examples for the requirement is shown in Fig. 9.

## VI. Limitations and Challenges

There are some obstructions for authentication of intended Agent-Oriented Supply-Chain Management. We restrict the number of orders to two. We also restrict the number of activities to two and the contractors against those activities. A customer can send maximum two orders and demands for maximum two or minimum one activity. These limitations reduce the state space because the model generates a huge state space. The machines are used in our verification have limited resources for memory and speed. These limitations are also used due to limited memory of machine. The machine can crash during execution of query verification phase. We perform some computations on the machine with 4GB RAM, core i3(3rd Gen) Laptop.

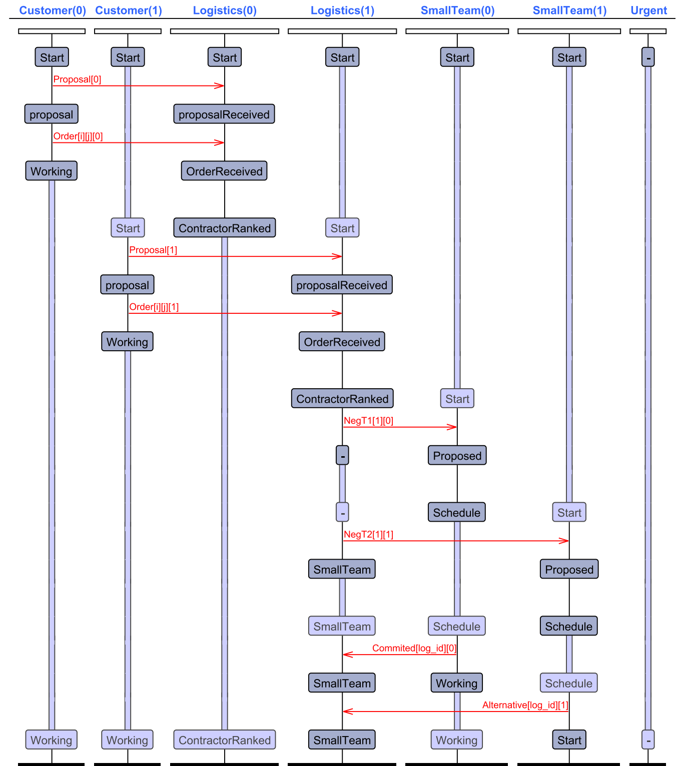


Fig. 8. Trace for requirement R1.

## VII. Conclusion

We formalized Agent-Oriented Supply-Chain Management as specified in [3] in UPPAAL model checker. We then formalized functional requirements of the architecture and verified them by model checking. Results show that the given architecture partially fulfills its functional requirements. Proof the results are presents in the form for message sequence charts. The given protocol is verified with limited number of logistic agents, orders and customers.

## References

[1] M. C. C. Douglas M. Lambert, "Issues in supply chain management," *Industrial Marketing Management*, vol. 29, p. 19, 2000.

[2] I. K. Nirupam Julka, Rajagopalan Srinivasan, "Agent-based supply chain management-1: framework," *Computers and Chemical Engineering*, vol. 26, p. 15, 2002.

[3] R. T. Mark S. Fox, Mihai Barbuceanu, "Agent-oriented supply-chain management," in *The International Journal of Flexible Manufacturing Systems, 12*. Kluwer Academic Publishers, 2000, pp. 165–188.

[4] C. Baier, J.-P. Katoen *et al.*, "Principles of model checking, vol. 26202649," *MIT Press Cambridge*, vol. 26, p. 58, 2008.

[5] P. R. D'Argenio, J.-P. Katoen, T. C. Ruys, and J. Tretmans, *The bounded retransmission protocol must be on time!* Springer, 1997.

[6] H. Lonn and P. Pettersson, "Formal verification of a tdma protocol start-up mechanism," in *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*. IEEE, 1997, pp. 235–242.

[7] P. Pettersson, *Modelling and verification of real-time systems using timed automata: theory and practice*. Citeseer, 1999.

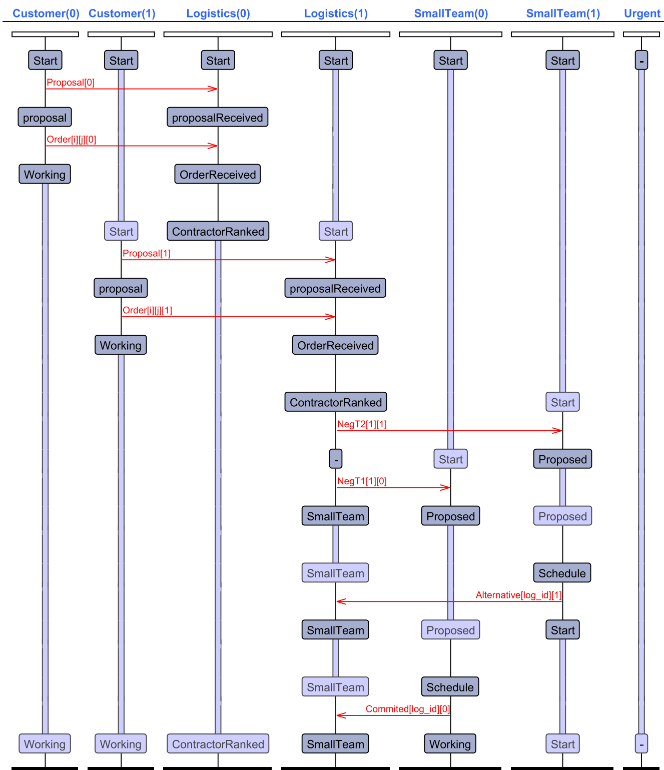


Fig. 9. Trace for requirement R7.

[8] W. Yi, P. Pettersson, and M. Daniels, “Automatic verification of real-time communicating systems by constraint- solving.” in *FORTE*, vol. 6. Citeseer, 1994, pp. 243–258.

[9] M. Atif, “Formal modeling and verification of distributed failure detectors,” *Faculty of Mathematics and Computer Science, TU/e*, vol. 10, 2011.