Formal specification and verification of a team formation protocol using TLA

Rajdeep Niyogi[1] and Amar Nath[2]

[1]Indian Institute of Technology Roorkee
[2]Sant Longowal Institute of Engineering and Technology

March 14, 2023

Abstract

Team formation in an environment where some relevant parameters are not known in advance is a challenging problem. Communicating automata and distributed algorithms have been used to describe protocols for team formation. A high-level specification provides a mathematical description of a protocol or a program. TLA $^+$ is a formal specification language designed to provide high-level specifications of concurrent and distributed systems. The associated model checker known as TLC is capable of model checking the TLA $^+$ specifications. Recently formal specification of a team formation protocol is given using TLA $^+$ when there is a single initiator (an agent or a robot), that initiates the team formation. Using TLA $^+$, we examine the formal specification for the multiple initiator situation and demonstrate that a composition technique can yield a single monolithic specification for the multiple initiator situation from the single initiator situation specification. We have used models of varying sizes, and the TLC model checker has confirmed that the protocol's specifications meet certain desired characteristics in each case.

EXTENDED CONFERENCE PAPER

Journal Section

Formal specification and verification of a team formation protocol using TLA$^+$

Rajdeep Niyogi[1*] | Amar Nath[2*]

[1]CSE, Indian Institute of Technology Roorkee, Roorkee, India-247667

[2]CSE, Sant Longowal Institute of Engineering & Technology (SLIET), Longowal, Punjab-India-148106

Correspondence
Rajdeep Niyogi, Indian Institute of Technology Roorkee, Roorkee, India-247667
Email: rajdeep.niyogi@cs.iitr.ac.in

Present address
[†]CSE, Sant Longowal Institute of Engineering & Technology (SLIET), Longowal, Punjab-India-148106

Funding information
na

Team formation in an environment where some relevant parameters are not known in advance is a challenging problem. Communicating automata and distributed algorithms have been used to describe protocols for team formation. A high-level specification provides a mathematical description of a protocol or a program. TLA$^+$ is a formal specification language designed to provide high-level specifications of concurrent and distributed systems. The associated model checker known as TLC is capable of model checking the TLA$^+$ specifications. Recently formal specification of a team formation protocol is given using TLA$^+$ when there is a single initiator (an agent or a robot), that initiates the team formation. Using TLA$^+$, we examine the formal specification for the multiple initiator situation and demonstrate that a composition technique can yield a single monolithic specification for the multiple initiator situation from the single initiator situation specification. We have used models of varying sizes, and the TLC model checker has confirmed that the protocol's specifications meet certain desired characteristics in each case.

KEYWORDS
Team formation, multi-agent system, formal specification, TLA$^+$, verification

[*] Equally contributing authors.

1 | INTRODUCTION

Task execution in a partially observable environment, where the agents (robots) are spatially distributed and no agent has a global view of the environment is a challenging problem. It has wide applications in search and rescue [1, 2, 3], space exploration [4], demining [5], where a team of autonomous robots is deployed since direct human intervention is impossible or impractical. Task execution is done by a team of agents (robots).

Collective robotic systems (CRS) [6], a team of autonomous robots that cooperate to execute a task, deployed in real-world scenarios should work as expected. Improper operation of the CRS can have a negative impact on safety and may result in financial loss. For the CRS, which is designed to operate in a dynamic environment, it is imperative to detect potential misbehavior before deploying the system on a real robotic system. Formal verification [7] is a process of checking whether a design satisfies the requirements. Formal verification using model checking of the distributed approaches for task execution [3, 8] has been suggested in [9, 10]. However, while model checking is widely used to verify designs of tightly controlled software and hardware systems, it is not used for requirements analysis. Moreover, formal specification of team formation protocol in formal languages (e.g., [11, 12, 13, 14]) has only been recently studied in [15].

Team formation is initiated by an agent, that has found a task, by exchanging messages (explicit communication). Distributed approaches for such team formation under different assumptions have been suggested in [8, 3, 16, 17]. In centralized approaches [18, 19], a team is guaranteed to be formed since all the relevant parameters are known a priori. Distributed approaches are used when some parameters are not known in advance and they should be acquired at run-time. Thus in distributed approaches, whether a team would be formed is determined by the current situation of the robots and communication delays at run-time. In some instances, the robot initiating the team formation process would be successful in forming a team and unsuccessful in others. In this paper, we are interested in the successful executions of the underlying team formation protocol [8, 3, 16, 17] (those that result in team formation). We wish to understand and specify what is allowed to happen for such an execution, and whether the specification satisfies some desirable properties.

A high-level specification [20, 21] permits thinking of software above code level and of hardware above circuit level. The behavioral aspects of a program or protocol, also known as its functional or logical properties, are described in the specification. In essence, such a specification ought to specify what a program can do rather than how it is accomplished or implemented. We demonstrate in this paper how to specify the logical properties of algorithm executions that lead to team formation.

Formal specification languages, like Alloy [13], TLA$^+$ [14], VDM [11], Z [12], have been used to specify software requirements. A recent survey [22] gives an excellent comparison of the different formal specification methods. The authors [22] are of the opinion that selecting a particular formal specification language depends on several requirements. For instance, the primary requirements for specifying and model-checking the Amazon Web Services model include the ability to model complex concurrent and distributed systems, effective tool support, a short learning curve, the impact of the method on overall development time, and minimal training [22].

A notation for specifying a system as a collection of actions is provided by Lamport's formal specification language, TLA$^+$ [14]. It was made to describe high-level specifications of distributed and concurrent systems. The fact that logical formulas that are a combination of first-order predicate calculus and temporal logic using only the "always" operator are used to specify both the system and its desirable properties is what gives TLA$^+$ its elegance. TLA$^+$ is a language for writing mathematics. TLA$^+$ comes with a tool for computer-assisted verification, the TLC model checker, that can verify any TLA$^+$ specification. The combination of TLA$^+$ and the TLC model checker provides a powerful tool for system specification and verification. TLA$^+$ is widely used in academia and industry, e.g., to verify data consistency in cloud applications. There is a TLA$^+$ textbook and a repository of learning material.

For our work, we need a formal specification language that can model distributed systems, supports tools well, and is easy to use. TLA$^+$ perfectly meets our requirements. Therefore, we chose TLA$^+$ and the TLC model checker for specifying and verifying the requirements and desirable properties of every successful execution of the team formation algorithms [8, 3, 16, 17].

In our earlier work [15], formal specification of the team formation protocol is discussed for one initiator using TLA$^+$, which is given in Section 5. The specific contributions in this paper are given below:

- We first show how a single monolithic specification for the multiple initiator situation can be obtained from the specification of the single initiator situation (Section 6.1).
- We give a generic model of the team formation protocol for $k > 1$ initiators, and illustrate for $k = 2$ (Section 6.2).
- We validate the specification for the multiple initiator situation with varying-sized models using the TLC model checker (Section 7).

The remaining part of the paper is organized as follows. Related work is discussed in Section 2. The team formation protocol is given in Section 3. A brief discussion of TLA$^+$ is given in Section 4. Conclusions are given in Section 8.

2 | STATE OF THE ART AND RELATED WORK

State transition systems are mostly used to specify the behavior of agents (or robots), and temporal logic formulas are used to specify desirable properties, according to a recent survey on formal specification and verification of autonomous robotic systems [23]. We talk about some work on robot teams.

STOKLAIM[24], a stochastic version of KLAIM [25], is a formal specification language used to specify the behaviors of robots that transport an object together [6]. The goal, obstacles, and robots are all described in detail at the lowest level in the specification. In KLAIM, behaviors are expressed as processes. Some processes that are taken into account in [6] include: a procedure that informs each robot of the obstacles nearby and periodically updates the robots' positions in accordance with their instructions. The stochastic logic MoSL[24] is used to specify the relevant properties, and for the formulas of the logic quantitative analysis for STOKLAIM specifications is done with the model checking tool SAM [24, 26]. In [6] team formation is not taken into account.

The SPIN model checker [29] has been used to verify each robot's individual safety property in the domestic assistant robot model [27, 28]. SPIN makes it possible to verify qualitative properties that do not require an explicit time or probability. Swarm robotics applications [31, 32, 33] have utilized the model checker known as PRISM (probabilistic symbolic model checker) [30]. Quantitative property verification is made possible by PRISM.

The aggregation algorithm is modeled with discrete-time Markov chains (DTMCs) [31], and probabilistic computation tree logic (PCTL*) is used to specify the characteristics of swarm behavior. The foraging scenario is modeled using DTMCs [32, 33], and PCTL is used to specify safety properties. In [8] a model based on communicating automata is used to capture robot behavior.

In [3, 8, 16, 17] distributed approaches for collaborative task execution are suggested based on different assumptions, depending on whether or not the team size is known in advance. Formal verification using model checking has been suggested in [9, 10] when the team size is known in advance. PRISM model checker is used in [9] and PCTL is used to describe quantitative properties. SPIN model checker is used in [10], and Promela, the input language for SPIN, is used to model the algorithm. LTL (linear temporal logic) is used to describe the properties.

In [15] it was shown that the behavior of the system comprising of the agents or robots and its desirable properties can be described as a single temporal logic formula using the language TLA$^+$. This is unlike SPIN model checking where the system is modeled using one language and the properties to be verified are described in another language. Moreover, the TLA$^+$ based specification allows for handling the different situations depending on the knowledge of the team size in an elegant manner (see the discussions on the condition $C1$ in Section 5.2). Changing this condition as per the requirement, we can address different situations, for example, team size is known, team size is unknown, and also the auction protocol.

3 | TEAM FORMATION PROTOCOL

The process of forming a team is first described. Take a look at a closed environment represented as a grid of size $m \times m$. The grid has some objects in some of its cells. A few robots are moving through the grid. The robots are responsible for working together to get rid of these objects from the grid. If a robot is able to remove an object by itself it does so. Otherwise, it seeks help from the other robots that are within its vicinity. The only way the robots can communicate is through messages. It is assumed in [3, 8, 16, 17] that the communication infrastructure is lossless and message delay is finite.

The works [3, 8, 16, 17] consider collaborative task execution in a dynamic environment where all the agents associated with the execution should be present at the location of the task. In [3, 8, 16, 17] distributed approaches for collaborative task execution are suggested based on different assumptions. In [3, 8] team size is known in advance, and each agent in a team should have a set of skills that is a superset of the set of skills required for a task. Suppose that a robot A has found an object O at cell (*x*,*y*) that requires two other robots that have the capability of pushing a heavy object (any object whose weight is, say, above 50 kg). If A is able to form a team, then the number of other suitable robots available nearby A should be at least two.

Unlike [3, 8], a generic situation is considered in [16, 17], where the combined set of skills of the agents of a team should be a superset of the skills required for a task. Thus the team size is not known in advance; rather determined at run-time in [16, 17]. Suppose that a robot A has found an object O at cell (*x*,*y*). When a team is transporting the object, the following skills are required: pushing, grabbing, and obstacle avoidance. The robot A has the capability of pushing but not the other skills. Thus A needs to form a team whose combined skill set has at least these three skills. In any case, robot A needs to communicate with other robots.

The team formation protocol for message exchange among the robots is now discussed. In this protocol, communication is assumed to be lossless and the message delay is arbitrary but finite. The robot that initiates the team formation process is referred to as the "initiator" (say A) and the other robots that participate in the process are referred to as "non-initiators." Assume that B,C,D are the three other robots that are available near A. Each robot possesses at least one of the three capabilities necessary to complete a task. A request message with the task's specifics (name, location, capabilities) is *broadcast* by A.

Since there is no loss of messages, the robots B,C,D will all receive the request message because they are close to A. Now B,C,D send willing messages to A containing information about their current location, cost, and capabilities. When A gets these willing messages, it chooses the best team it can. Let's say that A rejects D and chooses B and C. Now, A *notifies* the non-initiators of its decision by sending confirm to B,C and notRequired to D. A robot cannot send a message more than once under this protocol.

It is assumed that the robots will be in various states at various points during the execution of the protocol. For instance, when it broadcasts, the initiator is in the ready state. Before receiving the request message, all non-initiators are in the idle state. Their states change to promise as soon as they receive the request message. The states of `B, C` which were notified, change to busy, while `D` which was notified, changes to idle. In parallel, `A` makes its state busy.

The process of forming a team does not guarantee that one will always emerge. Here is an example of such a scenario. After broadcasting the request message, the initiator waits a certain amount of time, Δ, and records all willing messages received within that time frame. It ignores any willing messages that arrive after Δ. Team formation would fail if the initiator did not receive a sufficient number of willing messages within Δ. We are interested in those executions (runs) of the underlying team formation protocol [3, 8, 16, 17] for which team formation is successful. Multiple initiators can initiate their team formation process simultaneously, where the tasks for the initiators are different. If a non-initiator receives multiple request messages from multiple initiators, it responds to exactly one initiator's request and ignores the rest.

In [15] a formal specification of the team formation protocol is discussed for one initiator. The specification is contained in a module *TeamFormation*, given at the end of Section 7. In this paper, we consider the specification for multiple initiators. If there are multiple initiators, we first create an instance of the module *TeamFormation* for each initiator with suitable substitutions, and then compose these instances to obtain a single monolithic specification.

4 | TLA$^+$: A BRIEF INTRODUCTION

The concepts that would be helpful for comprehending the model presented in Section 5.1 are briefly discussed. For a detailed description, we refer to [21, 14]. The keywords of TLA$^+$ are written in uppercase sans serif font. A state is defined as an assignment of values to all the variables. Any infinite sequence of states is a behavior. A set of possible behaviors, which are the ones that signify the system's proper operation, is what is meant by "system specification."

The possible initial values of all the variables can be specified using the predicate $Init$. The relationship between their possible values in the next state and their current values are specified by the next-state relation $Next$. A step is a pair of successive states $\langle currentstate, nextstate \rangle$. As a result, $Next$ specifies how the variables' values change at each step. If v indicates a value in the current state, then v' indicates a value in the following state. The formula $Next$ has both primed and unprimed variables. Additionally, the variables whose values have not changed are identified by the keyword UNCHANGED. As a result, writing UNCHANGED v is equivalent to writing $v' = v$. A formula like this is known as an action that is either true or false of a step.

A TLA$^+$ specification is given as a single formula. This formula must assert two things about a behavior: (i) the formula $Init$ is satisfied in the initial state, and (ii) each of its steps satisfies $Next$. We express (i) as the formula $Init$, which we interpret as a statement about behaviors to mean that the initial state satisfies $Init$. To express (ii), we use the temporal-logic operator Box (always), which is the *only* temporal operator in TLA$^+$. The temporal formula $\Box\ P$ asserts that formula P is always true. So $\Box\ Next$ is the assertion that $Next$ is true for every step in the behavior. Thus, $Init \wedge \Box\ Next$ is true of a behavior if and only if the initial state satisfies $Init$ and every step satisfies $Next$.

A stuttering step is a step that doesn't change one or more variables. $[Next]_v$ means $Next \vee (v' = v)$. A "safety specification" in TLA$^+$ takes the form $Init \wedge \Box[Next]_{vars}$, where $vars$ is a tuple of specification variables.

A state function is an expression having only variables and constants; it cannot have prime or $\Box$. A Boolean-valued state function is called a state predicate. An invariant Inv of a specification $Spec$ is a state predicate when $Spec \implies \Box Inv$ is a theorem. A temporal formula satisfied by any behavior is called a theorem. This formula expresses that any behavior that satisfies $Spec$ also satisfies $\Box Inv$. TLA$^+$ is an untyped language, where every value is a set. An expression like $v \in Nat$ assigns a type to the variable v; in this case, Nat is the set of natural numbers. Formally, a variable v has type T in $Spec$ iff $v \in T$ is an invariant of $Spec$.

5 | SPECIFICATION OF THE TEAM FORMATION PROTOCOL: SINGLE INITIATOR

The specification of the team formation protocol for multiple initiators, given in the next Section, is constructed from the specification for the single initiator situation. Thus we present a step-by-step model of the team formation protocol for a single initiator in this section. The reasoning that led to the specification is discussed here. The content of this section is borrowed from [15]. The complete specification involving one initiator is contained in the module *TeamFormation*, given at the end of Section 7.

5.1 | Abstraction

We distinguish the robots based on their roles in the team formation process. There is an initiator and a set of non-initiators. An initiator is a robot that initiates the team formation process. A non-initiator is a robot whose willing message, in response to the initiator's request message, is received by the initiator within a predefined time period.

The initiator's request message is received, in general, by some of the agents present in the environment. The agents who did not receive the request message cannot participate in the team formation. The agents who received the request message but none of their skills match with that of the task cannot also participate in the team formation. The agents who received the request message and at least one of their skills match with that of the task send willing messages to the initiator. However, not all the willing messages may be received by the initiator within a predefined time period. Only those agents whose willing messages are received by the initiator within a predefined time period are considered for the team formation and such agents are referred to as non-initiators. The set of non-initiators do not change throughout the execution of the team formation protocol. Let *NonInitiator* denote the set of non-initiators. This is a constant parameter, one whose value remains the same at all the states. The statement

CONSTANT *NonInitiator*

declares *NonInitiator* to be a constant.

Now we have an initiator and a set of non-initiators that are fixed throughout the execution of the protocol. What changes during the protocol execution is the behavior of these agents. So in order to capture the behaviors, we model the agents using records. The record of the initiator has the fields *state* (to denote the current state), *member* (to denote the other team members i.e., the selected non-initiators), and *nonmember* (to denote the non-members, i.e., the rejected non-initiators). The record of a non-initiator has the fields *state* (to denote the current state), *sent*, and *recd* (to denote the last message sent and received respectively). So we have a variable $initiator$, a record, for the initiator. We have a variable $Data$, an array, where $Data[p]$ is a record for a non-initiator p. The statement

VARIABLES *initiator*, *Data* declares *initiator* and *Data* as variables.

The sets of records of the initiator and a non-initiator are defined as

initiatorRec $\triangleq$

[$state$: {"ready", "busy"},

member : SUBSET *NonInitiator*,

nonmember : SUBSET *NonInitiator*]

nonInitRec $\triangleq$

[$state$: {"idle", "promise", "busy"},

$sent$: {"willing", "nil"},

$recd$: {"request", "confirm", "notRequired", "nil"}]

Types of variables

The types of *initiator* and *Data* are given by the expressions:

$initiator \in initiatorRec$ and

$Data \in [NonInitiator \rightarrow nonInitRec]$

Recall that a variable v has type T iff $v \in T$ is an invariant of $Spec$. The variables $initiator$ and $Data$ have types $initiatorRec$ and $[NonInitiator \rightarrow nonInitRec]$ respectively, since both the expressions are type invariants of $Spec$. A detailed discussion of the meaning of these expressions is given in the subsection 5.3 where we describe the type invariant $TypeOK$.

Initial State

A state is the assignment of values to every variable. The variable $initiator$ is a record with fields $state$, $member$, and $nonmember$. In the initial state, the fields of $initiator$ take on the values:

$initiator.state =$ "ready", $initiator.member = initiator.nonmember = \{\}$.

The value of $initiator$ in the initial state is written as:

$initiator = [state \mapsto$ "ready", $member \mapsto \{\}, nonmember \mapsto \{\}]$

Similarly, the value of $Data$ in the initial state is written as:

$Data = [p \in NonInitiator \mapsto$

$[state \mapsto$ "idle", $sent \mapsto$ "nil", $recd \mapsto$ "nil"$]]$

So the record corresponding to a non-initiator p is written as:

$Data[p] = [state \mapsto$ "idle", $sent \mapsto$ "nil", $recd \mapsto$ "nil"$]$

In the following, some expressions would be used multiple times. For the sake of clarity, we define the following.

$InitialInitiator \triangleq$

$[state \mapsto$ "ready", $member \mapsto \{\}, nonmember \mapsto \{\}]$

$InitialNoninit \triangleq [state \mapsto$ "idle", $sent \mapsto$ "nil", $recd \mapsto$ "nil"$]$

$AfterReceiveNoninit \triangleq [state \mapsto$ "promise", $sent \mapsto$ "willing",

$recd \mapsto$ "request"$]$

$Init$, the predicate describing the initial state, is defined as the conjunction of the values of the two variables *initiator* and *Data*.

$$\boxed{\begin{aligned} Init \triangleq\ & \land\ initiator = InitialInitiator \\ & \land\ Data = [p \in NonInitiator \mapsto InitialNoninit] \end{aligned}}$$

In general, a TLA$^+$ definition like

$$\begin{aligned} D \triangleq\ & \land\ c_1 \\ & \ \vdots \\ & \land\ c_k \end{aligned}$$

is equivalent to $D \triangleq\ c_1 \land \ldots \land c_k$ (the logical AND operation).

5.2 | Actions

A specification describes what is allowed to happen in a state. This means what are the actions that are allowed at a state. Now at the initial state, the only thing that can happen is the initiator broadcast a request message. So we have an action $BroadcastRequest$.

1. $BroadcastRequest$

At the initial state, the action is enabled. The state that results from the action is one in which both the fields of the non-initiators change (*state*, *recd*) and none of the fields of the initiator change. The enabling conditions are provided by the definition's first two conjuncts. The last two conjuncts provide the conditions for the resulting state. The action is formally defined as

$$\boxed{\begin{array}{l} BroadcastRequest \triangleq \\ \land\ initiator = InitialInitiator \\ \land\ Data = [p \in NonInitiator \mapsto InitialNoninit] \\ \land\ Data' = [p \in NonInitiator \mapsto \\ [Data[p] \ \text{EXCEPT } !.state = \text{“promise"}, \ !.recd = \text{“request"}]] \\ \land\ \text{UNCHANGED } initiator \end{array}}$$

$[Data[p]$ EXCEPT $!.state = \text{“promise"}$, $!.recd = \text{“request"}]]$ is the new record obtained by changing only the fields *state* and *recd* of the record in the previous state; the value of the other fields remain same.

The value of the predicate $Init$ is true at s_0. Thus if the action $BroadcastRequest$ is performed at the initial state s_0, then the resulting state is s_1.

$$s_0 \xrightarrow{BroadcastRequest} s_1$$

The values of the variables at s_1 are:

$initiator = InitialInitiator$

$Data = [p \in NonInitiator \mapsto$

$[state \mapsto \text{“promise"}, \ sent \mapsto \text{“nil"}, \ recd \mapsto \text{“request"}]]$

For a team to be formed, the initiator must receive some willing messages. So we have an action $ReceiveWilling$.

2. $ReceiveWilling$

When the fields of a non-initiator are *state* = "promise" and *recd* = "request" and the initiator is in the initial state, the action is enabled. The non-initiators sent the messages because the initiator receives the willing messages. Therefore, the action has the effect of changing the non-initiator's *sent* field to "willing", while the initiator's value remains unchanged. The enabling conditions are provided by the definition's first two conjuncts. The conditions for the state that will result are provided by the two final conjuncts. The action is formally defined as

$$\boxed{\begin{array}{l} ReceiveWilling \triangleq \\ \land\ initiator = InitialInitiator \\ \land\ Data = [p \in NonInitiator \mapsto \\ [state \mapsto \text{“promise"}, \ sent \mapsto \text{“nil"}, \ recd \mapsto \text{“request"}]] \\ \land\ Data' = [p \in NonInitiator \mapsto \\ [Data[p] \ \text{EXCEPT } !.sent = \text{“willing"}]] \\ \land\ \text{UNCHANGED } initiator \end{array}}$$

The action $ReceiveWilling$ is enabled at the state s_1. Thus if $ReceiveWilling$ is performed at s_1, then the resulting state is s_2.

$$s_1 \xrightarrow{ReceiveWilling} s_2$$

The values of the variables at s_2 are:

$initiator = InitialInitiator$

$Data = [p \in NonInitiator \mapsto$

$[state \mapsto \text{“promise"}, sent \mapsto \text{“willing"}, recd \mapsto \text{“request"}]]$

3. *SelectTeam*

The state where the value of $initiator$ corresponds to the value in the initial state and the value of $Data$ to the record after receiving the “willing" messages is the state in which the action is enabled. The enabling condition is provided by the definition's first two conjuncts. The members of the team are chosen by the initiator (the fields $member$ and $nonmember$ are updated in the third conjunction of the definition). We want to specify that a team is selected, not how. This is expressed by taking a nonempty subset of $NonInitiator$ (the third conjunct of the definition). The equation $\exists q \in S : F$ asserts that the set S contains a q that satisfies the formula F; The value of $Data$ remains unchanged (the fourth conjunct of the definition), and q is a local variable whose scope is defined by the formula F. The action is formally defined as

$$\boxed{\begin{array}{l} SelectTeam \triangleq \\ \land\ initiator = InitialInitiator \\ \land\ Data = [p \in NonInitiator \mapsto AfterReceiveNoninit] \\ \land\ \exists v \in \text{SUBSET } NonInitiator : \land\ C1 \\ \qquad\qquad\qquad\qquad\qquad\qquad \land\ C2 \\ \land\ \text{UNCHANGED } Data \end{array}}$$

$C1 = Cardinality(v) > 0$

$C2 = initiator' = [initiator \text{ EXCEPT } !.member = v, !.nonmember = NonInitiator \setminus v]$

In order to obtain a nonempty subset of $NonInitiator$, a local variable v is used. The value of the field $member$ is set to the value of v, and the value of the field $nonmember$ is set to the value of the remaining elements of $NonInitiator$ (\ is the set difference operator) in the above definition.

The condition $C1$ can be designed in accordance with the need. For instance, when $C1 = Cardinality(v) > 0 \land v \neq NonInitiator$, not all non-initiators are chosen. We obtain the situation in which the team size is known in advance if $Cardinality(v) = k$, where k is declared as a constant parameter to denote the number of additional members required by the initiator. For an auction protocol set $k = 1$.

In the previous actions, the initiator sent and received messages from non-initiators, demonstrating the interaction between various system components. Dissimilar to these activities, $SelectTeam$ is an inner activity that isn't because of any message. The action $SelectTeam$ is enabled at the state s_2. Thus if $SelectTeam$ is performed at s_2, then the resulting state is s_3.

$$s_2 \xrightarrow{SelectTeam} s_3$$

The values of the variables at s_3 are:

$initiator = [state \mapsto \text{“ready"}, member \mapsto v, nonmember \mapsto NonInitiator \setminus v]$

$Data[p] = AfterReceiveNoninit$

4. *Notify*

When $initiator.state$ = “ready" (first conjunct of the definition), $initiator.member \neq \{\}$ (second conjunct of the definition), and the value of $Data$ corresponds to the record after obtaining the “willing" messages (third conjunct of the definition), the action is enabled. All elements of $initiator.member$ should have the values of the fields $state$ equal to “busy", $recd$ equal to “confirm", and all elements of $initiator.nonmember$ should have the values of the fields $state$ equal to “idle", $recd$ equal to “notRequired", $sent$ equal to “nil" (fourth conjunct of the definition). The IF/THEN/ELSE construct is used to express this. The state of the initiator also becomes “busy" (last conjunct of the definition). The action *Notify* is formally defined as

$$\boxed{\begin{array}{l} Notify \triangleq \\ \land\ initiator.state = \text{“ready"} \\ \land\ initiator.member \neq \{\} \\ \land\ Data = [p \in NonInitiator \mapsto AfterReceiveNoninit] \\ \land\ Data' = [p \in NonInitiator \mapsto \\ \text{IF } p \in initiator.member \text{ THEN} \\ [Data[p]\ \text{ EXCEPT } !.state = \text{“busy"},\ !.recd = \text{“confirm"}] \\ \text{ELSE } [state \mapsto \text{“idle"},\ sent \mapsto \text{“nil"},\ recd \mapsto \text{“notRequired"}]] \\ \land\ initiator' = [initiator\ \text{ EXCEPT } !.state = \text{“busy"}] \end{array}}$$

The action $Notify$ is enabled at the state s_3. Thus if $Notify$ is performed at s_3, then the resulting state is s_4.

$s_3 \xrightarrow{Notify} s_4$

The values of the variables at s_4 are:

$initiator = [state \mapsto \text{“busy"},\ member \mapsto v,\ nonmember \mapsto NonInitiator \backslash v]$

$Data[p] = [state \mapsto \text{“busy"},\ sent \mapsto \text{“willing"},\ recd \mapsto \text{“confirm"}]$ where $p \in initiator.member$

$Data[p] = [state \mapsto \text{“idle"},\ sent \mapsto \text{“nil"},\ recd \mapsto \text{“notRequired"}]$ where

$p \in initiator.nonmember$

At a state, only one action is enabled, and all enabled actions are distinct, according to the definitions above. A finite sequence of states is produced as a result of the finite number of actions. Proper execution in TLA$^+$ is defined as an infinite sequence of states. Therefore, a stuttering step known as *StutStep* repeats the sequence's final state infinitely many times while maintaining the variables' unchanged values.

$$\boxed{\begin{array}{l} StutStep \triangleq \land\ initiator.member \neq \{\} \\ \qquad\qquad\quad \land\ \text{UNCHANGED } \langle Data, initiator \rangle \end{array}}$$

So the sequence of states and actions of the protocol would be:

$s_0 \xrightarrow{BroadcastRequest} s_1 \xrightarrow{ReceiveWilling} s_2 \xrightarrow{SelectTeam} s_3 \xrightarrow{Notify} s_4 \xrightarrow{StutStep} s_4, \ldots$

There are 5 distinct states $s_0, \ldots, s_4$. The final state s_4 repeats by virtue of *StutStep*.

Thus the next step relation *Next* is defined as follows.

$Next \triangleq BroadcastRequest \lor ReceiveWilling \lor SelectTeam \lor Notify \lor StutStep$

Thus the specification is defined as:

$Spec \triangleq Init \land \Box[Next]_{vars}$ where $vars \triangleq \langle initiator, Data \rangle$.

That the specification $Spec$ is complete (meaning that it specifies what should occur for any possible action), has been verified by TLC model checker.

5.3 | Predicates

We define the state predicate $TypeOK$, which describes the types of the variables, as:

$$\boxed{\begin{aligned} TypeOK \triangleq\ & \land\ initiator \in initiatorRec \\ & \land\ Data \in [NonInitiator \rightarrow nonInitRec] \end{aligned}}$$

The type invariant $TypeOK$ asserts that (i) the value of $initiator$ is an element of the set of records $initiatorRec$, and (ii) the value of $Data$ is an element of the set of functions f with $f[x] \in nonInitRec$ for $x \in NonInitiator$ (i.e., the value of $Data[x]$ is an element of the set of records $nonInitRec$).

The invariance of the state predicate $Consistency$, which is the conjunction of two conditions, demonstrates the protocol's correctness. The first asserts that the protocol is not in an inconsistent final state in which the initiator is either in a busy state and the team members are in another state, or the team members are busy and the initiator is in another state. The second asserts that the cardinality of the set of team members and non-members is less than or equal to that of the set of non-initiators as a whole.

$$\boxed{\begin{aligned} & Consistency \triangleq \\ & \land\ (initiator.state = \text{“busy"}) \equiv C3 \land C4 \\ & \land\ Cardinality(initiator.member) + \\ & Cardinality(initiator.nonmember) \leq Cardinality(NonInitiator) \end{aligned}}$$

$C3 = Cardinality(initiator.member) > 0$

$C4 = \forall p \in initiator.member : Data[p].state = \text{“busy"}$

We have a theorem asserting the invariance of $TypeOK$ and $Consistency$. The theorem is verified by TLC model checker.

THEOREM $Spec \implies \Box(TypeOK \land Consistency)$

where $Spec \triangleq Init \land \Box[Next]_{vars}$

6 | SPECIFICATION OF THE TEAM FORMATION PROTOCOL: MULTIPLE INITIATORS

In the previous section, we described the design of the team formation protocol for one initiator. In this section, we describe the design of a specification of the protocol for multiple initiators. For illustration, we consider two initiators, denoted by i, j, that are completely independent of one another.

6.1 | Composition of individual specifications

We give a monolithic specification TTF (two team formations) obtained by the composition of individual specifications. For the details of the composition of specifications, we refer to [14]. Such a specification should have the form:

$$\begin{aligned} & TTF \triangleq \\ & \qquad \land\ Init \\ & \qquad \land\ \Box[Next]_{vars} \end{aligned}$$

$iSpec \triangleq iInit \land \Box[iNext]_{ivars}$ be the specification of the protocol when there is only one initiator i as described in Section 5.

$jSpec \triangleq jInit \land \Box[jNext]_{jvars}$ be the specification of the protocol when there is only one initiator j as described in Section 5.

$TTF \triangleq iSpec \land jSpec$ be the specification of the protocol when there are two initiators i, j.

By substituting the definitions of $iSpec, jSpec$ we get

$$\begin{aligned}
&TTF \triangleq\\
&\qquad \land\, iInit \land \Box[iNext]_{ivars}\\
&\qquad \land\, jInit \land \Box[jNext]_{jvars}
\end{aligned}$$

By rearranging the terms we get

$$\begin{aligned}
&TTF \triangleq\\
&\qquad \land\, iInit \land jInit\\
&\qquad \land\, \Box[iNext]_{ivars} \land \Box[jNext]_{jvars}
\end{aligned}$$

Let $Init \triangleq iInit \land jInit$. By substitution we get

$$\begin{aligned}
&TTF \triangleq\\
&\qquad \land\, Init\\
&\qquad \land\, \Box[iNext]_{ivars} \land \Box[jNext]_{jvars}
\end{aligned}$$

Since $\Box P \land \Box Q \equiv \Box(P \land Q)$, so we get

$$\begin{aligned}
&TTF \triangleq\\
&\qquad \land\, Init\\
&\qquad \land\, \Box([iNext]_{ivars} \land [jNext]_{jvars})
\end{aligned}$$

Since $[Next]_v = Next \lor (v' = v)$, so by substitution we get

$$\begin{aligned}
&TTF \triangleq\\
&\qquad \land\, Init\\
&\qquad \land\, \Box(\land\, iNext \lor ivars' = ivars\\
&\qquad\qquad \land\, jNext \lor jvars' = jvars)
\end{aligned}$$

Rewriting the terms $(a \lor b) \land (c \lor d)$ as $(a \land c) \lor (a \land d) \lor (b \land c) \lor (b \land d)$ we get

$$\begin{aligned}
&TTF \triangleq\\
&\quad \land\, Init\\
&\quad \land\, \Box(\lor\, iNext \land jNext\\
&\qquad\quad \lor\, iNext \land jvars' = jvars\\
&\qquad\quad \lor\, jNext \land ivars' = ivars\\
&\qquad\quad \lor\, ivars' = ivars \land jvars' = jvars)
\end{aligned}$$

Since $Next \lor (v' = v) = [Next]_v$, so by substitution we get

$$\begin{aligned}
&TTF \triangleq\\
&\quad \land\, Init\\
&\quad \land\, \Box[\lor\; iNext \land jNext\\
&\qquad\quad \lor\; iNext \land jvars' = jvars\\
&\qquad\quad \lor\; jNext \land ivars' = ivars]_{vars}
\end{aligned}$$

Thus $TTF \triangleq Init \land \Box[Next]_{vars}$ is a monolithic specification like $iSpec, jSpec$ where the next step relation $Next$ is

$Next \triangleq$

$\vee\ iNext \wedge jNext$

$\vee\ iNext \wedge jvars' = jvars$

$\vee\ jNext \wedge ivars' = ivars$

In $Next$, the first disjunct $iNext \wedge jNext$ allows simultaneous actions of the two initiators. If we do not want simultaneous actions, then we simply remove this disjunct that results in only interleaved actions (the second and third disjuncts).

Thus, if we want the two initiators to perform simultaneous actions, then a new action is defined that is a conjunction of individual actions. For example,

$BroadcastRequest \triangleq iBroadcastRequest \wedge jBroadcastRequest$

Interleaved action, on the other hand, is represented by the disjunction of individual actions. In the specification TTF we consider both interleaved as well as simultaneous actions.

6.2 | Modeling Two Team Formation

In order to illustrate the design of the specification we consider two initiators. However, as we shall see at the end of this subsection, this model is indeed generic, i.e., we can easily write a specification involving several initiators.

The team formation protocol developed in [8, 17, 16] can be invoked simultaneously by multiple initiators. However, an agent cannot participate in more than one team formation process simultaneously. So we have two disjoint sets of non-initiators $iNonInitiator, jNonInitiator$ corresponding to i, j respectively. The parameters are declared in TLA$^+$ as:

CONSTANTS $iNonInitiator, jNonInitiator$

VARIABLES *iInitiator, iData, jInitiator, jData*

The following statement says that it is assumed that the sets $iNonInitiator$ and $jNonInitiator$ are disjoint.

ASSUME $iNonInitiator\ \cap jNonInitiator = \{\}$

We use the definitions of the $TeamFormation$ module for specifying the actions of the initiators i, j. So we obtain two instances of that module with suitable substitutions of the parameters. The first of these instances is given by the statement

$InitiatorI \triangleq$ INSTANCE $TeamFormation$ WITH

$NonInitiator \leftarrow iNonInitiator,\ Data \leftarrow iData,\ initiator \leftarrow iInitiator$

where the instance $InitiatorI$ is obtained by replacing the parameters of the

$TeamFormation$ module with the corresponding parameters of the current module; $NonInitiator$ replaced with $iNonInitiator$, $initiator$ with $iInitiator$, and $Data$ with $iData$. The other instance is given by the statement

$InitiatorJ \triangleq$ INSTANCE $TeamFormation$ WITH

$NonInitiator \leftarrow jNonInitiator,\ Data \leftarrow jData,\ initiator \leftarrow jInitiator$

Let P denote a statement that has the form

$$P \triangleq \text{INSTANCE } M \text{ WITH } E_1 \leftarrow e_1, \ldots, E_n \leftarrow e_n$$

Let σ be any symbol defined in the module M. The statement $P!\sigma$ has the same meaning in the current module as σ had in module M with the replacements $E_1 \leftarrow e_1, \ldots, E_n \leftarrow e_n$.

The initial state predicate $Init$ is defined as:

$Init \triangleq$

  $\wedge\ InitiatorI!Init$

  $\wedge\ InitiatorJ!Init$

It may be noted that the usage of $Init$ in the conjuncts and the definition name does not create any conflict. The above definition says that the meaning of the predicate $Init$ in the current module is the conjunction of the meaning of $Init$ in the module $TeamFormation$ with replacements given in $InitiatorI, InitiatorJ$.

6.2.1 | Actions

1. $BroadcastRequest$

We want to specify that both the initiators simultaneously broadcast. So we define $BroadcastRequest$ as a conjunction of the individual actions of the initiators, i.e., $iBroadcastRequest$ and $jBroadcastRequest$. Now the action $iBroadcastRequest$ of initiator i can be defined using the definition of the action $BroadcastRequest$ in the $TeamFormation$ module as

$iBroadcastRequest \triangleq$

  $\wedge\ InitiatorI!BroadcastRequest$

  $\wedge\ \text{UNCHANGED}\langle iInitiator, jInitiator\rangle$

$jBroadcastRequest \triangleq$

  $\wedge\ InitiatorJ!BroadcastRequest$

  $\wedge\ \text{UNCHANGED}\langle iInitiator, jInitiator\rangle$

$BroadcastRequest \triangleq$

  $\wedge\ iBroadcastRequest$

  $\wedge\ jBroadcastRequest$

2. $ReceiveWilling$

This action may be performed either simultaneously or in an interleaved manner. We give the specification when the individual actions are interleaved.

$iReceiveWilling \triangleq$

  $\wedge\ InitiatorI!ReceiveWilling$

  $\wedge\ \text{UNCHANGED}\langle iInitiator, jInitiator, jData\rangle$

$jReceiveWilling \triangleq$

$\land\ InitiatorJ!ReceiveWilling$
$\land\ \text{UNCHANGED}\langle iInitiator, jInitiator, iData\rangle$

$ReceiveWilling \triangleq$
$\lor\ iReceiveWilling$
$\lor\ jReceiveWilling$

3. $SelectTeam$

This action may be performed either simultaneously or in an interleaved manner. We give the specification when the individual actions are interleaved.

$iSelectTeam \triangleq$
$\land\ InitiatorI!SelectTeam$
$\land\ \text{UNCHANGED}\langle iData, jData, jInitiator\rangle$
$jSelectTeam \triangleq$
$\land\ InitiatorJ!SelectTeam$
$\land\ \text{UNCHANGED}\langle iData, jData, iInitiator\rangle$

$SelectTeam \triangleq\ \lor\ iSelectTeam$
$\lor\ jSelectTeam$

4. $Notify$

This action may be performed either simultaneously or in an interleaved manner. We give the specification when the individual actions are interleaved.

$iNotify \triangleq$
$\land\ InitiatorI!Notify$
$\land\ \text{UNCHANGED}\langle jData, jInitiator\rangle$

$jNotify \triangleq$
$\land\ InitiatorJ!Notify$
$\land\ \text{UNCHANGED}\langle iData, iInitiator\rangle$

$Notify \triangleq\ \lor\ iNotify$
$\lor\ jNotify$

5. $StutStep \triangleq$
$\land\ iNonInitiator \neq \emptyset$
$\land\ iNonInitiator \neq \emptyset$
$\land\ \text{UNCHANGED}\langle iData, jData, iInitiator, jInitiator\rangle$

$Next \triangleq$
$\lor\ BroadcastRequest$
$\lor\ ReceiveWillling$
$\lor\ SelectTeam$
$\lor\ Notify$

$\quad \lor\ StutStep$

6.2.2 | Invariants

$$\begin{aligned} TypeOK \triangleq\ & \land\ InitiatorI!TypeOK \\ & \land\ InitiatorJ!TypeOK \end{aligned}$$

$$\begin{aligned} Consistency \triangleq\ & \land\ InitiatorI!Consistency \\ & \land\ InitiatorJ!Consistency \end{aligned}$$

We now have the complete specification: $TTF \triangleq Init \land \Box[Next]_{vars}$

We have a theorem asserting the invariance of $TypeOK$ and $Consistency$. The theorem is verified by the TLC model checker.

THEOREM $TTF \implies \Box(TypeOK \land Consistency)$

Thus, the composed specification involving k initiators ($k > 1$) can be easily obtained by reusing the definitions involving only one initiator in the module *TeamFormation*, given at the end of Section 7. The specification for two initiators is given in module *TeamFormationTwoInitiator*, given at the end of Section 7.

7 | VERIFICATION USING TLC MODEL CHECKER

7.1 | Implementation Results

Microsoft Visual Studio Code with TLC 2 Version 2.16 is used for writing the TLA$^+$ specifications and verification of the codes on a computer having the configuration: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz, 16.0 GB (15.7 GB usable) RAM, 64-bit operating system, x64-based processor, Windows 10 Pro, version 21H1.

Model 1: We consider 2 initiators where $|iNonInitiator| = 4$ and $|jNonInitiator| = 5$. The output obtained by TLC is given below. No error has been found. 6051 states generated, 2049 distinct, 0 states left on queue. The depth of the complete state graph search is 8. TLC finished the model checking in 568ms. The details of states for each action are given below. Success means no error has been found, Diameter is the depth of the complete state graph search, found indicates the number of states generated, Distinct indicates the number of distinct states, and the entry for Queue indicates 0 states left on queue. The Coverage shows the states generated for each action and $Init$. If we remove *StutStep*, TLC reports an error “deadlock reached".

```
-------Output of TLC for Model 1--------
java.exe -cp c:\Users\Admin\.vscode\extensions\alygin.vscode-tlaplus-
1.5.4\tools\tla2tools.jar -XX:+UseParallelGC tlc2.TLC
TeamFormationTwoInitiator.tla -tool -modelcheck -coverage 1 -config
TeamFormationTwoInitiator.cfg
TLC2 Version 2.16 (rev: cdddf55)
Running breadth-first search Model-Checking with fp 49 and seed -
3505808484348599909 with 1 worker on 8 cores with 227MB heap and 64MB offheap
memory (Windows 10 10.0 x86, Oracle Corporation 1.8.0_261 x86, MSBDiskFPSet,
DiskStateQueue).
Starting SANY...
Parsing file C:\Users\Admin\Dropbox\My PC (DESKTOP-
VIOGDQD)\Desktop\myCodesTLA\TeamFormationTwoInitiator.tla
Parsing file C:\Users\Admin\AppData\Local\Temp\Naturals.tla
Parsing file C:\Users\Admin\AppData\Local\Temp\FiniteSets.tla
Parsing file C:\Users\Admin\Dropbox\My PC (DESKTOP-
VIOGDQD)\Desktop\myCodesTLA\TeamFormation.tla
Parsing file C:\Users\Admin\AppData\Local\Temp\Sequences.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module FiniteSets
Semantic processing of module TeamFormation
Semantic processing of module TeamFormationTwoInitiator
SANY finished.
Starting...
Computing initial states...
Finished computing initial states:
1 distinct state generated Model checking completed.
No error has been found.
Estimates of the probability that TLC did not check all reachable states because
two distinct states had the same fingerprint: calculated (optimistic): val = 4.4E-13
6051 states generated, 2049 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 8.
The average outdegree of the complete state graph is 1 (minimum is 0, the maximum
31 and the 95th percentile is 2). Finished in 568ms
```

```
------More Output of TLC for Model 1: details of states-----
Status
Checking TeamFormationTwoInitiator.tla / TeamFormationTwoInitiator.cfg
Success Fingerprint collision probability: 4.4E-13
Start: 12:18:15 end: 12:18:15
States
Time------Diameter----Found----Distinct---Queue
00:00:00   8          6051      2049       0
Coverage
Module--------------------Action-------------------Total----------Distinct
TeamFormationTwoInitiator Init---------------------1----------------1
TeamFormationTwoInitiator BroadcastRequest---------1----------------1
TeamFormationTwoInitiator iReceiveWilling----------64---------------1
TeamFormationTwoInitiator jReceiveWilling----------32---------------2
TeamFormationTwoInitiator iSelectTeam--------------960--------------30
TeamFormationTwoInitiator jSelectTeam--------------992--------------527
TeamFormationTwoInitiator iNotify------------------960--------------495
TeamFormationTwoInitiator jNotify------------------992--------------992
TeamFormationTwoInitiator StutStep-----------------2049-------------0
```

Model 2: We consider 2 initiators where $|iNonInitiator| = 7$ and $|jNonInitiator| = 7$. No error has been found. 196099 states generated, 65537 distinct, 0 states felt on queue. The depth of the complete state graph search is 8. TLC finished the model checking in 3522ms. The details of states for each action are given below.

```
----More Output of TLC for Model 2: details of states-----
Checking TeamFormationTwoInitiator.tla / TeamFormationTwoInitiator.cfg
Success Fingerprint collision probability: 4.6E-10
Start: 12:21:22 end: 12:21:25
States
Time-----Diameter----Found---Distinct----Queue
00:00:00   0           1        1           1
00:00:03   8        196099    65537         0
Coverage
Module--------------------Action-----------Total------Distinct
TeamFormationTwoInitiator Init--------------1----------1
TeamFormationTwoInitiator BroadcastRequest--1----------1
TeamFormationTwoInitiator iReceiveWilling---256--------1
TeamFormationTwoInitiator jReceiveWilling---256--------2
TeamFormationTwoInitiator iSelectTeam-------32512------254
TeamFormationTwoInitiator jSelectTeam-------32512------16383
TeamFormationTwoInitiator iNotify-----------32512------16383
TeamFormationTwoInitiator jNotify-----------32512------32512
TeamFormationTwoInitiator StutStep----------65537-------0
```

Model 3: We consider 3 initiators where $|iNonInitiator| = |jNonInitiator| = |kNonInitiator| = 4$. No error has been found. 25538 states generated, 8289 distinct states found, 0 states left on queue. The depth of the complete state graph search is 9. TLC finished the model checking in 950ms. The details of states for each action are given below.

```
----More Output of TLC for Model 3: details of states-----
Checking  MultiTeamFormation.tla  / MultiTeamFormation.cfg
Success Fingerprint collision probability: 7.8E-12
Start: 12:08:14 end: 12:08:15
States
Time-----Diameter----Found----Distinct---Queue
00:00:00   9          25538     8289        0
Coverage
Module-------------Action-----------------Total--------Distinct
MultiTeamFormation Init---------------------1--------------1
MultiTeamFormation BroadcastRequest---------1--------------1
MultiTeamFormation iReceiveWilling----------289------------1
MultiTeamFormation jReceiveWilling----------289------------2
MultiTeamFormation kReceiveWilling----------289------------4
MultiTeamFormation iSelectTeam--------------4335-----------60
MultiTeamFormation jSelectTeam--------------4335-----------510
MultiTeamFormation kSelectTeam--------------4335-----------4335
MultiTeamFormation Notify-------------------3375-----------3375
MultiTeamFormation StutStep-----------------8289------------0
```

Model 4: We consider 3 initiators where $|iNonInitiator| = |jNonInitiator| = |kNonInitiator| = 5$. No error has been found.200066 states generated, 65729 distinct states found, and 0 states left on queue. The depth of the complete state graph search is 9. TLC finished the model checking in 4052ms. The details of states for each action are given below.

```
-----More Output of TLC for  Model 4: details of states-----
Status
Checking MultiTeamFormation.tla / MultiTeamFormation.cfg
Success Fingerprint collision probability: 4.8E-10
Start: 12:14:08 end: 12:14:12
States
Time-----Diameter-----Found----Distinct----Queue
00:00:00   0           1         1          1
00:00:03   9         200066    65729        0
Coverage
Module-------------Action-----------------Total--------Distinct
MultiTeamFormation Init--------------------1-----------1
MultiTeamFormation BroadcastRequest--------1-----------1
MultiTeamFormation iReceiveWilling---------1089--------1
MultiTeamFormation jReceiveWilling---------1089--------2
MultiTeamFormation kReceiveWilling---------1089--------4
MultiTeamFormation iSelectTeam-------------33759-------124
MultiTeamFormation jSelectTeam-------------33759-------2046
MultiTeamFormation kSelectTeam-------------33759-------33759
MultiTeamFormation Notify------------------29791-------29791
MultiTeamFormation StutStep----------------65729-------0
```

The TLA$^+$ specification for one initiator:

```
---- MODULE TeamFormation----
EXTENDS Naturals, FiniteSets
CONSTANT NonInitiator \* set of non-initiators
VARIABLES initiator, Data \* initiator, Data[p] are records
vars == <<initiator, Data>>
CN == Cardinality(NonInitiator)
Cm == Cardinality(initiator.member)
Cnm == Cardinality(initiator.nonmember)
InitialInitiator == [state |-> "ready", member |->{}, nonmember |-> {}]
InitialNoninit == [state |-> "idle", sent |-> "nil", recd |-> "nil"]
AfterReceiveNoninit == [state |-> "promise",
sent |-> "willing", recd |-> "request"]
nonmemberRec == [state |-> "idle", sent |-> "nil",
                 recd |-> "notRequired"]
nonInitRec ==
       [state : {"idle", "busy", "promise"},
        sent : {"willing", "nil"},
        recd : {"request", "confirm", "notRequired", "nil"}]
initiatorRec ==
        [state : {"ready", "busy"},
       member : SUBSET NonInitiator,
       nonmember: SUBSET NonInitiator]
TypeOK == /\ initiator \in initiatorRec
          /\ Data \in [NonInitiator |-> nonInitRec]
Init ==  /\ initiator = InitialInitiator
         /\ Data = [p \in NonInitiator |-> InitialNoninit]
-------------------------------------
BroadcastRequest ==
/\ Data = [p \in ÂăNonInitiator |-> InitialNoninit]
/\ initiator = InitialInitiator
/\ Data' = [p \in NonInitiator |-> [Data[p] EXCEPT !.state = "promise", !.recd = "
                                              request"]]
/\ UNCHANGED initiator
ReceiveWilling ==
/\ initiator = InitialInitiator
/\ Data = [p \in NonInitiator |-> [state |-> "promise", sent |-> "nil", recd |->"
                                              request"]]
/\ Data' = [p \in NonInitiator |-> [Data[p] EXCEPT !.sent = "willing"]]
/\ UNCHANGED  initiator
SelectTeam ==
/\ initiator = InitialInitiator
/\ Data = [p \in NonInitiator |-> AfterReceiveNoninit]
/\ \E v \in SUBSET NonInitiator : /\ Cardinality(v) > 0
/\ initiator' = [initiator EXCEPT !.member = v, !.nonmember = NonInitiator \ v]
/\ UNCHANGED Data
Notify ==
/\ initiator.state = "ready"
/\ initiator.member # {}
/\ Data = [p \in NonInitiator |-> AfterReceiveNoninit]
/\ Data' = [p \in NonInitiator |-> IF p \in
initiator.member THEN
[Data[p] EXCEPT !.state = "busy", !.recd = "confirm"]
 ELSE nonmemberRec]
/\ initiator' = [initiator EXCEPT !.state = "busy"]
 StutStep ==
 /\ initiator.member # {}
 /\ UNCHANGED <<Data, initiator>>
Next ==  \/ BroadcastRequest  \* initiator broadcasts a request message
         \/ ReceiveWilling \* initiator receives the willing messages from the non-
                                          initiators
         \/ SelectTeam \* initiator selects the other team members and nonmembers
         \/ Notify \* initiator notifies the members and nonmembers
         \/ StutStep \* stuttering step
Spec == Init /\ [][Next]_vars \* The complete specification
Consistency == /\ (initiator.state = "busy") <=> /\ (Cm > 0)
                                                 /\ (\A p \in initiator.member : Data[p
                                          ].state = "busy")
              /\ Cm + Cnm <= CN
------------------------------------------
THEOREM Spec =>  [](TypeOK /\ Consistency) \* An invariant of Spec
==============================
```

The TLA$^+$ specification for two initiators:

```
---- MODULE TeamFormationTwoInitiator----
EXTENDS Naturals, FiniteSets
CONSTANTS iNonInitiator, jNonInitiator \* set of non-initiators
VARIABLES iInitiator, iData, jData, jInitiator
\* iInitiator, iData[p] are records
InitiatorI == INSTANCE TeamFormation WITH NonInitiator <- iNonInitiator, Data <- iData,
                                           initiator <- iInitiator
InitiatorJ == INSTANCE TeamFormation WITH NonInitiator <- jNonInitiator, Data <- jData,
                                           initiator <- jInitiator
vars == <<iInitiator, iData, jData, jInitiator>>
ASSUME iNonInitiator \cap jNonInitiator = {}
iInitiatorRec == InitiatorI!initiatorRec
jInitiatorRec == InitiatorJ!initiatorRec
TypeOK == /\ InitiatorI!TypeOK
          /\ InitiatorJ!TypeOK
Consistency == /\ InitiatorI!Consistency
               /\ InitiatorJ!Consistency
Init == /\ InitiatorI!Init
        /\ InitiatorJ!Init
-------------------------------------
iBroadcastRequest == /\ InitiatorI!BroadcastRequest
                     /\ UNCHANGED <<iInitiator, jInitiator>>
jBroadcastRequest == /\ InitiatorJ!BroadcastRequest
                     /\ UNCHANGED <<iInitiator, jInitiator>>
BroadcastRequest == /\ iBroadcastRequest
                    /\ jBroadcastRequest
iReceiveWilling == /\ InitiatorI!ReceiveWilling
                   /\ UNCHANGED <<iInitiator, jInitiator, jData>>
```

```
jReceiveWilling == /\ InitiatorJ!ReceiveWilling
                   /\ UNCHANGED <<iInitiator, jInitiator, iData>>
ReceiveWilling == \/ iReceiveWilling
                  \/ jReceiveWilling
iSelectTeam == /\ InitiatorI!SelectTeam
               /\ UNCHANGED <<iData, jData, jInitiator>>
jSelectTeam == /\ InitiatorJ!SelectTeam
               /\ UNCHANGED <<iData, jData, iInitiator>>
SelectTeam == \/ iSelectTeam
              \/ jSelectTeam
iNotify == /\ InitiatorI!Notify
           /\ UNCHANGED <<jData, jInitiator>>
jNotify == /\ InitiatorJ!Notify
           /\ UNCHANGED <<iData, iInitiator>>
Notify == \/ iNotify
          \/ jNotify
StutStep ==  /\ iNonInitiator # {}
             /\ jNonInitiator # {}
             /\ UNCHANGED <<iData, jData, iInitiator, jInitiator>>
Next == \/ BroadcastRequest  \* initiator broadcasts a request message
        \/ ReceiveWilling \* initiator receives the willing messages from the non-initiators
        \/ SelectTeam \* initiator selects the other team members and nonmembers
        \/ Notify \* initiator notifies the members and nonmembers
        \/ StutStep \* stuttering step
TTF == Init /\ [][Next]_vars \* The complete specification
------------------------------------------
THEOREM TTF => [](TypeOK /\ Consistency) \* An invariant of Spec
==============================
```

8 | CONCLUSIONS

Protocols for the formation of teams have been described using distributed algorithms and communicating automata. TLA$^+$ has recently been used to provide formal specifications for a team formation protocol when there is only one initiator. In this paper, we considered the formal specification for the multiple initiator situation using TLA$^+$. We showed that for the multiple initiator situation, a single monolithic specification can be obtained from the specification of the single initiator situation by a composition technique. We used models of varying sizes and in each case, the TLC model checker verified that the specifications meet some desirable properties of the protocol in each case. The outcomes demonstrate that the protocol's behavior is consistent with expectations. A mathematical description of a protocol or program can be found in a high-level specification. As a result, through the use of formal specification language, we have provided a comprehensive understanding of the team formation procedure for the typical circumstance involving multiple initiators.

TLA$^+$ provided satisfactory levels of abstraction and expressiveness for our work. When compared to the Promela model [10], the logical properties of the protocol executions are better understood by using the TLA$^+$ specification, which is presented as a single formula.

references

[1] S. Tadokoro, H. Kitano, T. Takahashi, I. Noda, H. Matsubara, A. Shinjoh, T. Koto, I. Takeuchi, H. Takahashi, F. Matsuno, M. Hatayama, J. Nobe, and S. Shimada. 2000. The robocup-rescue project: A robotic approach to the disaster mitigation problem. In ICRA. 4089-4094.

[2] T. Gunn and J. Anderson. 2015. Dynamic heterogeneous team formation for robotic urban search and rescue. J. Comput. System Sci. 81, 3 (2015), 553-567

[3] A. Nath, A. R. Arun, and R. Niyogi. 2019. A distributed approach for road clearance with multi-robot in urban search and rescue environment. International Journal of Intelligent Robotics and Applications 3, 4 (2019), 392-406

[4] N. Agmon and P. Stone. 2012. Leading ad hoc agents in joint action settings with multiple teammates. In AAMAS. 341-348

[5] M. Hemapala, V. Belotti, R. Michelini, and R. Razzoli. 2009. Humanitarian demining: path planning and remote robotic sweeping. Industrial Robot: An International Journal 36, 2 (2009), 146-156.

[6] Gjondrekaj, E., Loreti, M., Pugliese, R., Tiezzi, F., Pinciroli, C., Brambilla, M., Birattari, M., and Dorigo, M. Towards a Formal Verification Methodology for Collective Robotic Systems. In: 14th International Conference on Formal Engineering Methods, pp. 54–70, 2012.

[7] C. Baier and J.-P. Katoen. 2008. Principles of model checking. MIT press.

[8] Nath, A., Arun, A.R, and Niyogi, R. An approach for task execution in dynamic multirobot environment. In: Australasian Joint Conference on Artificial Intelligence, pp. 71–76, 2018.

[9] A. Nath and R. Niyogi. 2020. Formal verification of a distributed algorithm for task execution. In 20th Int. Conf. on Computational Science and its Applications (ICCSA), LNCS 12253 pp 120-131, 2020.

[10] A. Nath and R. Niyogi. 2021. Formal modeling, verification, and analysis of a distributed task execution algorithm. In 35th IEEE Int. Conf. on Advanced Information Networking and Applications (AINA), LNNS 225, pp 370-382, 2021.

[11] Jones, C. B. Systematic Software Development using VDM. Prentice Hall. 1990.

[12] Spivey, M. The Z Notation, Prentice Hall International. 1992.

[13] Jackson, D. Software Abstractions: Logic, Language, and Analysis. MIT press. 2006.

[14] Lamport, L. Specifying Systems. The TLA$^+$ Language and Tools for Hardware and Software Engineers, Addison-Wesley. 2002.

[15] Niyogi, R. Formal specification of a team formation protocol. 36th IEEE Int. Conf. on Advanced Information Networking and Applications (AINA), LNNS 451, pp 301-313, 2022.

[16] Nath, A., AR, A., and Niyogi, R. A distributed approach for autonomous cooperative transportation in a dynamic multi-robot environment. In: The 35th ACM Symposium on Applied Computing (ACM-SAC), pp. 792-799, 2020.

[17] Nath, A., Arun, A. R., and Niyogi, R. DMTF: A Distributed Algorithm for Multi-team Formation. In: 12th International Conference on Agents and Artificial Intelligence (ICAART), vol. 1, pp. 152-160, 2020.

[18] T. Lappas, K. Liu, and E. Terzi. 2009. Finding a team of experts in social networks. In Proceedings of the 15th international conference on Knowledge discovery and data mining. ACM, 467-476

[19] H. J. GutiÃl'rrez, A. S. Astudillo, P. Ballesteros-PÃl'rez, and A. Candia-VÃl'jar. 2016. The multiple team formation problem using sociometry. Computers and Operations Research 75 (2016), 150-162.

[20] Batson, B and Lamport, L. High-Level Specifications: Lessons from Industry. In: First International Symposium on Formal Methods for Components and Objects, 2002.

[21] Lamport, L. The TLA Home Page, http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html

[22] A. Mashkoor, F. Kossak, and A. Egyed. 2018. Evaluating the suitability of state based formal methods for industrial deployment. Software: Practice and Experience 48 (2018), 2350-2379.

[23] Luckcuck, M., Farrell, M., Dennis, A. L., Dixon, C. and Fisher, M. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. ACM Computing Surveys 52(5), 2019.

[24] De Nicola, R., Katoen, J., Latella, D., Loreti, M., and Massink, M. Model checking mobile stochastic logic. Theor. Comput. Sci. 382(1), 42-70, 2007.

[25] De Nicola, R., Ferrari, G., Pugliese, R. KLAIM: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. Softw. Eng. 24(5), 315-330, 1998.

[26] Loreti, M. SAM: Stochastic Analyser for Mobility, http://rap.dsi.unifi.it/SAM

[27] Webster, M., Dixon, C., Fisher, M., Salem, M., Saunders, J., Koay, L. K., Dautenhahn, K. and Saez-Pons, J. Toward reliable autonomous robotic assistants through formal verification: A case study. Trans. of Human-Machine Systems, 46(2), 186-196, 2016.

[28] Webster, M., Dixon, C., Fisher, M., Salem, M. Saunders, J., Koay, L. K. and Dautenhahn, K. Formal verification of an autonomous personal robotic assistant. In: AAAI FVHMS, pp. 74-79, 2014.

[29] Holzmann, G. Spin model checker, the primer and reference manual. Addison Wesley Professional. 2003.

[30] Kwiatkowska, M., Norman, G., and Parker, D. Prism: Probabilistic symbolic model checker, In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, pp. 200–204, 2002.

[31] Brambilla, M., Pinciroli, C., Birattari, M. and Dorigo, M. Property-driven design for swarm robotics. In: International Conference on Autonomous Agents and Multiagent Systems, pp. 139–146, 2012.

[32] Konur, S., Dixon, C., and Fisher, M. Analysing robot swarm behavior via probabilistic model checking. Robotics and Autonomous Systems 60(2), 199-213, 2012.

[33] Konur, S., Dixon, C., and Fisher, M. Formal verification of probabilistic swarm behaviours. Swarm Intelligence 440-447, 2010.

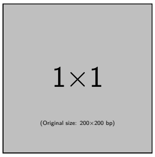
[34] Lamport, L. A Simple Approach to Specifying Concurrent Systems. Commun. ACM 32(1), 32-45, 1989.

[35] Lamport, L. Specifying Concurrent Program Modules. ACM Trans. Program. Lang. Syst. 5(2), 190-222, 1983.

[36] Gerkey, B. P. and Mataric, M. J. Sold!: Auction methods for multirobot coordination. IEEE Trans. on Robotics and Automation, 18(5), 758-768, 2002.

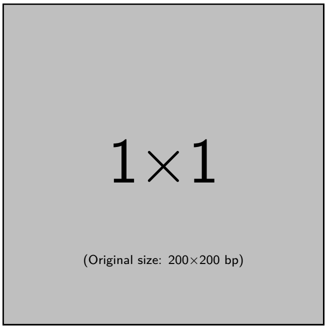
[37] Kong, Y., Zhang, M., and Ye, D. An auction-based approach for group task allocation in an open network environment. The Computer Journal, 59(3), 403-422, 2015.

[38] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker. M. and Deardeuff, M. How Amazon web services uses formal methods. Commun. ACM 58(4), 66–73, 2015.



A. One Please check with the journal's author guidelines whether author biographies are required. They are usually only included for review-type articles, and typically require photos and brief biographies (up to 75 words) for each author.

GRAPHICAL ABSTRACT



Please check the journal’s author guildines for whether a graphical abstract, key points, new findings, or other items are required for display in the Table of Contents.