Egyptian Computer Science Journal Vol.45.No.1 January 2021, ISSN-1110-2586

Automated Rapid Prototyping of TUG Specifications for Implementing Atomic Read/Write Shared Memory in Mobile Ad Hoc Networks

Reham.A.Shihata
PhD in Computer Science, EL Menoufia University –Egypt. Software Consultant
.in Egyptian Syndicate of Programmers & Scientists
anwerreham@yahoo.com

Abstract: A Rapid prototyping has been used for exploring vague user requirements in the front-end of the software life cycle. Automated rapid prototyping may reduce cost of prototyping and the time of developing it .One automated rapid prototyping technique is the direct execution of a specification. Direct execution of a specification has the benefits of quick construction of the prototype, direct support for formal specification, and quick response to the specification changes. However existing formal specification languages still have difficulties in specifying software systems such as non functional behavior of the systems. For non-executable formal specification languages, a prototype may be derived from the specification via software transformations. This approach to rapid prototyping uses a formal specification language to automatically generate a prototype in Prolog via a set of software transformation rules. Because there is a direct correspondence between the language and Prolog, the transformation is mechanical and straight forward. Specifiers can concentrate on generating the prototype without the distraction of transforming one notation into another. This formal specification language may not provide enough abstractions for provide enough abstraction for prototyping some particular features of systems. Therefore, this approach is designed to support the derived prototype to be extended or modified in a modular manner. The specification is written in modules in terms of the language patterns that support module independence, the prototype is then derived in a modular way that supports the ease of modifications to the prototype. The software transformation rules used for the derivation of prototypes in Prolog are presented. In this paper, this specification is applied on the implementation for atomic object Read/Write shared memory in mobile ad hoc networks (note*:* a CRS refinement on each invocation of the main application of the paper is presented by uppercase letters, but in the same application the based associating abstract atomic objects with certain geographic locations and services are represented by lowercase letters).

Keywords: Rapid Prototyping, TUG Specification, Prolog, Quorum Systems, CRS Refinement, Mobile Ad Hoc Networks.

I. Introduction

Tree Unified with Grammar (TUG) was developed to support a system to be developed through an integration of conventional software development, operational specification, rapid prototyping via software transformations, software reuse, and analysis of specifications and programs via testing and proofs. The language integrates various software development paradigms into a coherent whole to fit specific needs of developing organizations. This language improves the reusability of formal specifications in the following ways [1][2][3]:

(1) A developer can run a TUG specification as a prototype to study its behavior due to its executability at the front-end of the software life cycle
(2) A developer can easily write a parametric program corresponding to its parametric.

Rapid prototyping has been used for exploring vague user requirements in the front-end of the software life cycle. When the vague user requirements are understood, the prototype is thrown away. Therefore, a throwaway prototype should be built quickly and cheaply. Manual rapid prototyping methods aren't cost effective for building throwaway prototypes. Automated rapid prototyping may reduce the cost of prototyping and the time of developing it. One automated rapid prototyping technique is the direct execution of a specification. Direct execution of a specification has the benefits of quick construction of the prototype, direct support for formal specifications, and quick response to the specification changes. However, existing formal specification languages still have difficulties in specifying software systems such as non-functional behavior of the system for non-executable formal specification languages a prototype may be derived from the specification via software transformations]. Automatic rapid prototyping has its benefits; however, it also has the problems. The use of a prototype depends on the capability of its specification. In addition, re-derivation of prototypes from scratch may not be efficient if the specification is large. The approach to rapid prototyping uses a formal specification language to automatically generate a prototype in prolog via a set of software transformation rules. Because there is a direct correspondence between the language and prolog, the transformation is mechanical and straight forward. Specifiers can concentrate on generating the prototype without the distraction of transforming one notation into another. In order to avoid the re-derivation of the entire prototype from scratch we allow the prototype to be updated only in response to the revised specification never the less, if there is a major change in the specification, a new prototype is suggested to be regenerated from scratch. Like other existing formal specification languages, our formal specification language may not provide enough abstractions for prototyping some particular features of systems. Therefore, this approach is designed to support the derived prototype to be extended or modified in a modular manner. The specification is written in modules in terms of the language patterns that support module independence, the prototype is then derived in a modular way that supports the ease of modifications to the prototype. The software transformation rules used for the derivation of prototypes in prolog are presented. TUG specification language (Tree Unified with Grammar) was developed to support a system to be developed through an integration of conventional software development, operational specification, and rapid prototyping via software transformations, software reuse, and analysis of specifications and programs via testing and proofs. The language integrates various software development paradigms into a

coherent whole to fit specific needs of developing organizations. The TUG specification language consists of 3 parts: a name part where the title with input/output parameters are given, an analysis part where the input data is defined, and an anatomy part where the output data is generated. The name part contains a module or scheme title with input/output parameters the input/output parameters are enclosed in parentheses. The analysis part contains the rules for analyzing the input data. To analyze the input data, Definite Clause Grammar DCGs are used to represent the rules to perform the syntax analysis. Each rule of a DCG expresses a possible form for a nonterminal, as a sequence of terminals with optional constraints on the terminals and nonterminal. Nonterminal nodes in uppercase indicate constituents. A terminal node in lowercase indicates a token that must occur in the input data. A terminal node can be a literal which is any string enclosed in a pair of quotes. A constraint wrapped in braces places the conditions such as type checking on a terminal node. This tree representation will be the input to the anatomy analysis part of the TUG specification. The idea of prototyping via Software transformations isn't new. However, as automatic rapid prototyping approach should avoid a proto- type to be rederived from scratch whenever there is a change in the specification. Also, the automated approach should allow developers to easily extend the functions of the prototype manually in case the specification language doesn't support abstractions for features needed for demonstration. A rapid prototyping approach via Software transformations is presented to achieve this goal. User requirement are first written into specification in TUG. The specification needs not necessarily be complete, precise and correct corresponding to the user requirements at the first attempt. However, the specification should comply with the syntax of the language in order to be further processed. Next, a prototype in prolog is automatically derived from the specification. Via software transformations. The prototype is then exercised by the specifier and the use to clarify the user requirements in the front- end of the Software life cycle [4][5][6].

II. The Geoquorum – Approach

In this paper the GeoQuorums approach has presented for implementing atomic read/write shared memory in mobile ad hoc networks. This approach is based on associating abstract atomic objects with certain geographic locations. It is assumed that the existence of Focal Points, geographic areas that are normally "populated" by mobile nodes. For example: a focal point may be a road Junction, a scenic observation point [2]. Mobile nodes that happen to populate a focal point participate in implementing a shared atomic object, using a replicated state machine approach. These objects, which are called focal point objects, are prone to occasional failures when the corresponding geographic areas are depopulated. The Geoquorums algorithm uses the fault-prone focal point objects to implement atomic read/write operations on a fault-tolerant virtual shared object. The Geoquorums algorithm uses a quorum- based strategy in which each quorum consists of a set of focal point objects. The quorums are used to maintain the consistency of the shared memory and to tolerate limited failures of the focal point objects, which may be caused by depopulation of the corresponding geographic areas. The mechanism for changing the set of quorums has presented, thus improving efficiency [7]. Overall, the new Geoquorums algorithm efficiently implements read/write operations in a highly dynamic, mobile network. In this chapter, a new approach to designing algorithms for mobile ad hoc networks is presented. An ad hoc network uses no pre-existing infrastructure, unlike cellular networks that depend on fixed, wired base stations. Instead, the network is formed by the mobile nodes themselves, which co-operate to route communication from sources to destinations. Ad hoc communication networks are by nature, highly dynamic. Mobile nodes are often small devices with limited energy that spontaneously join and leave the network. As a mobile node moves, the set of neighbors with which at can directly communicate may change completely. The nature of ad hoc networks makes it challenging to solve the standard problems encountered in mobile computing, such as location management using classical tools. The difficulties arise from the lack of a fixed infrastructure to serve as the backbone of the network. In this chapter developing a new approach that allows existing distributed algorithm to be adapted for highly dynamic ad hoc environments one such fundamental problem in distributed computing is implementing atomic read/ write shared memory [8]. Atomic memory is a basic service that facilitates the implementation of many higher level algorithms. For example: one might construct a location service by requiring each mobile node to periodically write its current location to the memory. Alternatively, a shared memory could be used to collect real – time statistics. The problem of implementing atomic read/write memory is divided into two parts; **first**, we define a static system model, the focal point object model that associates abstract objects with certain fixed geographic locales. The mobile nodes implement this model using a replicated state machine approach. In this way, the dynamic nature of the ad hoc network is masked by a static model. Moreover it should be noted that this approach can be applied to any dynamic network that has a geographic basis. **Second**, an algorithm is presented to implement read/write atomic memory using the focal point object model. The implementation of the focal point object model depends on a set of physical regions, known as focal points [9].The mobile nodes within a focal point cooperate to simulate a single virtual object, known as a focal point object. Each focal point supports a local broadcast service, LBcast which provides reliable, totally ordered broadcast. This service allows each node in the focal point to communicate reliably with every operation completely. The focal broadcast service is used to implement a type of a replicated state machine, one that tolerates joins and leaves of mobile nodes. If a focal point becomes depopulated, then the associated focal point object fails. (Note that it doesn't matter how a focal point becomes depopulated, be it as a result of mobile nodes failing, leaving the area, going to sleep. etc. Any depopulation results in the focal point failing). The Geoquorums algorithm implements an atomic read/write memory algorithm on top of the geographic abstraction, that is, on top of the focal point object model. Nodes implementing the atomic memory use a Geocast service to communicate with the focal point objects. In order to achieve fault tolerance and availability, the algorithm replicates the read/write shared memory at a number of focal point objects. In order to maintain consistency, accessing the shared memory requires updating certain sets of focal points known as quorums. An important aspect of this approach is that the members of our quorums are focal point

objects, not mobile nodes. The algorithm uses two sets of quorums (I) **get-quorums** (II) **put- quorums** with property that every get-quorum intersects every put-quorum. There is no requirement that put-quorums intersect other put-quorums, or get-quorums intersect other get-quorums. The use of quorums allows the algorithm to tolerate the failure of a limited number of focal point objects. This algorithm uses a Global Position System (GPS) time service, allowing it to process write operations using a single phase, prior single-phase write algorithm made other strong assumptions, for example: relying either on synchrony or single writers. This algorithm guarantees that all read operations complete within two phases, but allows for some reads to be completed using a single phase: the atomic memory algorithm flags the completion of a previous read or write operation to avoid using additional phases, and propagates this information to various focal paint objects (see fig.1). As far as we know, this is an improvement on previous quorum based algorithms. For performance reasons, at different times it may be desirable to use different times it may be desirable to use different sets of get quorums and put-quorums [10]. For example: during intervals when there are many more read operations than write operations, it may be preferable to use smaller get- quorums that are well distributed, and larger put-quorums that are sparsely distributed. In this case a client can rapidly communicate with a get-quorum while communicating with a put – quorum may be slow. If the operational statistics change, it may be useful to reverse the situation.

A. Mathematical Notation for Geoquorums Approach

- I the totally- ordered set of node identifiers.
- $i_0 \in I$, a distinguished node identifier in I that is smaller than all order identifiers in I.
- S, the set of port identifiers, defined as $N^{>0} \times OP \times I$,
  Where OP= {get, put, confirm, recon- done}.
- O, the totally- ordered, finite set of focal point identifiers.
- T, the set of tags defined as $R^{\geq 0} \times I$.
- U, the set of operation identifiers, defined as $R^{\geq 0} \times S$.
- X, the set of memory locations for each $x \in X$:
  - V_x the set of values for x
  - $v_{0,x} \in V_x$, the initial value of X
- M, a totally-ordered set of configuration names
- $c_0 \in M$, a distinguished configuration in M that is smaller than all other names in M.
- C, totally- ordered set of configuration identifies, as defined as:
  $R^{\geq 0} \times I \times M$
- L, set of locations in the plane, defined as $R \times R$

Fig.1 Notations Used in the Geoquorums Algorithm.

B. Variable Types for Atomic Read/Write object in Geoquorum Approach for Mobile Ad Hoc Network

The specification of a variable type for a read/write object in geoquorum approach for mobile ad hoc network is presented and a read/write object has the following variable type (see fig .2).

Put/get variable type τ
State
Tag $\in$ T, initially$< 0.i_0>$
Value $\in$ V, initially v_0
Config-id $\in$ C, initially$< 0, i_0, c_0>$
Confirmed-set C T, initially Ø
Recon-ip, a Boolean, initially false
Operations
Put (new-tag, new-value, new-config-id)
If (new-tag> tag) then
Value ←new-value
Tag ← new-tag
If (new-config-id > config-id) then
Config-id ← new-config-id
Recon-ip ← true

Fig.2 A Read /Write Objects in the Application

C. Operation Manager

In this section the Operation Manger (OM) is presented, an algorithm built on the focal/point object Model. As the focal point Object Model contains two entities, focal point objects and Mobile nodes, two specifications is presented , on for the objects and one for the application running on the mobile nodes [11][12] .

1) Operation Manager Client: This automaton receives read, write, and recon requests from clients and manages quorum accesses to implement these operations (see fig .3). The Operation Manager (OM) is the collection of all the operation manager clients (OM_i, for all i in I).it is composed of the focal point objects, each of which is an atomic object with the put/get variable type.

Signature:
Input:
Write $(Val)_i$,val $\in$ V
read $(\ )_i$
recon $(cid)_I$, cid $\in$ C
respond (resp) $_{obj, P,}$ resp $\in$ responses (τ), obj $\in$ O, P = <*,*, i> $\in$ S
geo-update $(t,L)_i$, t $\in$ $R^{\geq 0}$, L $\in$ L
output:
write-ack $(\ )_i$
read-ack $(val)_i$,val $\in$ V
recon-ack $(cid)_i$, cid $\in$ C
invoke (inv) $_{obj, P}$, inv $\in$ invocations (τ),obj $\in$ O, P = <*,*,i> $\in$ S
Internal:
read-2 $(\ )_i$
recon-2 $(cid)_i$, cid $\in$ C
State:
Confirmed $\subseteq$ T, a set of tag ids, initially Ø
Conf-id $\in$ C, a configuration id, initially <0, i_0, c_0>
Recon- ip, a Boolean flag initially false
Clock $\in$ $R^{\geq 0}$, a time, initially 0
Ongoing - invocations $\subseteq$ O× S a set of objects and ports initially Ø
Current-port-number $\in$ $N^{>0}$, used to invoke objects, initially 1
Op, a record with the following components:
Type $\in$ {read, write, recon}, initially read
Phase $\in$ {{idle, get, put}, initially idle
Tag $\in$ T, initially <0, i_0>
Value $\in$ V, initially v_0
Recon-ip, a Boolean flag, initially false
Recon-conf-id $\in$ C, a configuration id, initially <0, i_0, c_0>
Acc $\subseteq$ 0, a set of data objects, initially Ø

Fig. 3 Operation Manager Client Signature and State for Node i in I, where τ is the Put/Get Variable Type.

D. Focal Point Emulator Overview

The focal point emulator implements the focal point object Model in an ad hoc mobile network. The nodes in a focal point (i.e. in the specified physical region) collaborate to implement a focal point object. They take advantage of the powerful LBcast service to implement a replicated state machine that tolerates nodes continually joining and leaving .This replicated state machine consistently maintains the state of the atomic object, ensuring that the invocations are performed in a consistent order at every mobile node [13]. In this section an algorithm is presented to implement the focal point object model. the algorithm allows mobile nodes moving in and out of focal points, communicating with distributed clients through the geocast service, to implement an atomic object (with port set q=s)corresponding to a particular focal point. We refer to this algorithm as the Focal Point Emulator (FPE).fig .4 contains the signature and state of the FPE .the code for the FPE client . The FPE client has three basic purposes. First, it ensures that each invocation receives at most one response (eliminating duplicates).Second, it abstracts away the geocast communication, providing a simple invoke/respond interface to the mobile node[14] [15]. Third, it provides each mobile node with multiple ports to the focal point object; the number of ports depends on the atomic object being implemented. The remaining code for the FPE server is in fig .8.When a node enters the focal point, it broadcasts a join-request message using the LBcast service and waits for a response. The other nodes in the focal point respond to a join-request by sending the current state of the simulated object using the LBcast service. As an optimization, to avoid unnecessary message traffic and collisions, if a node observes that someone else has already responded to a join-request, and then it does not respond. Once a node has received the response to its join-request, then it starts participating in the simulation, by becoming active. When a node receives a Geocast message containing an operation invocation, it resends it with the lbcast service to the focal point, thus causing the invocation to become ordered with respect to the other LBcast messages (which are join-request messages, responses to join requests, and operation invocations).since it is possible that a Geocast is received by more than one node in the focal point ,there is some bookkeeping to make sure that only one copy of the same invocation is actually processed by the nodes. There exists an optimization that if a node observes that an invocation has already been sent with LBcast service, then it does not do so. Active nodes keep track of operation invocations in the order in which they receive them over the LBcast service. Duplicates are discarded using the unique operation ids. The operations are performed on the simulated state in order. After each one, a Geocast is sent back to the invoking node with the response. Operations complete when the invoking node with the response. Operations complete when the invoking node remains in the same region as when it sent the invocation, allowing the geocast to find it. When a node leaves the focal point, it re-initializes its variables [16] [17].A subtle point is to decide when a node should start collecting invocations to be applied to its replica of

the object state. A node receives a snapshot of the state when it joins. However by the time the snapshot is received, it might be out of date, since there may have been some intervening messages from the LBcast service that have been received since the snapshot was sent. Therefore the joining node must record all the operation invocations that are broadcast after its join request was broadcast but before it received the snapshot .this is accomplished by having the joining node enter a "listening" state once it receives its own join request message; all invocations received when a node is in either the listening or the active state are recorded, and actual processing of the invocations can start once the node has received the snapshot and has the active status. A precondition for performing most of these actions that the node is in the relevant focal point. This property is covered in most cases by the integrity requirements of the LBcast and Geocast services, which imply that these actions only happen when the node is in the appropriate focal point[2] [18].

Signature:

Input

Geocast-rcv (< invoke, inv, oid, Loc>) $_{obj,i}$,inv $\in$ invocations, oid $\in$ U,loc $\in$ L (i.e. oid: object identifier , loc: location)

Lbcast-rcv (<Join-req, jid >) $_{obj,i}$, Jid $\in$ T (i.e. jid: join identifier)

Lbcast-rcv (<Join-ack, jid, v>) $_{obj,I}$, Jid $\in$ T, v$\in$V

Lbcast-rcv (<invoke, inv, oid, loc>) $_{obj,\ i}$, inv $\in$ invocations, oid $\in$U, loc $\in$L

Geo-update (l,t)$_{obj,I}$,l $\in$ L ,t $\in$ $R^{>0}$

Output:

Geocast (<response, resp, oid, loc>) $_{obj,i}$, resp $\in$Responses, oid$\in$U , loc$\in$L

Lbcast (< Join-req, Jid>) $_{obj,\ i}$, jid $\in$ T

Lbcast (< Join-ack, Jid, v>) $_{obj,\ i}$, jid $\in$ T, v$\in$ V

Lbcast (< invoke, inv, oid, loc>) $_{obj,\ i}$, inv $\in$ invocations, oid $\in$ U, loc $\in$ L

Internal:

Join ( ) $_{obj,i}$

Leave ( ) $_{obj,i}$

Simulate-op(inv)$_{obj,i}$, inv$\in$invocations.

State:

Fp-location $\in 2^L$, constant, locations defining the focal point under consideration

Clock $\in$ $R^{>0}$, the current time, initially 0, updated by the geosensor.

Location$\in$, L the current physical location, updater by the geosensor

Status $\in$ {idle, joining, listening, active}, initially active if node is in FP-location and idle otherwise.

Join- id $\in$ T, unique id for current join request, initially <0, i_0>.

Lbcast- queue, a queue of messages to be sent by the LBcast, initially Ø.

Geocast-queue, a queue of messages to be sent by the Geocast, initially Ø.

Answered-join-reqs set of ids of Join requests that have already been answered, initially Ø.

val $\in$ V, holds current value of the simulated atomic object, initially v_0.

Pending-ops, queue of operations waiting to be simulated, initially Ø.

Completed-ops, queue of operations that have been simulated, initially Ø.

Fig.4 FPE server signature and state for node i and object obj of variable type τ = <V, v_0, invocations, responses, δ >

III. A Specification in TUG

TUG specification language consists of 3 parts: a name part where the title with input/ output parameters is given, on analysis part where the input data is defined, and an anatomy part where the output data is generated. The name part contains a module or schema title with input/ output parameters are enclosed in parentheses. The analysis part contains the rules for analyzing the input data. To analyze the input data, Definite Clause Grammars (DCG_S) are used to represent the rules to perform the syntax analysis. Each rule of a DCG expresses a possible form for a non-terminal, as a sequence of terminals with optional constraints on the terminals and non-terminals. Non terminal nodes in uppercase indicate constituents. A terminal node in lowercase indicates a taken that must occur in the input data. A terminal node can be a literal which is any string enclosed in a pair of quotes. A constraint wrapped in braces places the conditions such as type checking indicates a taken that must occur in the input data. A terminal node can be a literal which is any string enclosed in a pair of quotes. A constraint wrapped in braces places the conditions such as type checking on a terminal node. Table I includes all operators used in the conditions. An input is parsed into a tree representation that takes the form of a prolog list with a node name acting as the relationship symbol of the input data. This tree representation will be the input to the anatomy analysis part of the TUG specification [19][20] [21].

IV. Rapid Prototyping Process via Software Transformations

The prototype serves as a basis for discussion to help the specifier and the user to read just the user requirements and specifications. Feedback from the user is obtained to decide whether the change is minor or major. If the change is minor, A Change Request Script (CRS) specifying the change is written to update the specification and the prototype. If a major change is needed, the specifier may rewrite the specification and rederive a new prototype from the start. A major change may involve the structure of the specification to be modified. This prototyping process continues until the requirements have been thoroughly exercised and the user is satisfied with the demonstrated behavior of the prototype. The results of the prototype evolution are a set of modular TUG specifications for the proposed system. In addition, a set of CRSs record the design decisions made during the transformations [22].There is no existing specification language that can support abstractions for all features of Software systems. Therefore, a specification language must make developers to easily extend and modify prototypes (see fig.5). To support easy modifications to the prototype, the TUG specification language was designed to support the construction of a specification in a structured manner with regular expression notations. The modules in a derived prototype from the specification can be easily located, modified, or extended in terms of these regular expression notations. The rapid prototyping approach using TUG can be incorporated into any Software development process. It is intended that each evolution of the specification that is synthesized by the specifier should be formally recorded using TUG, and that the prototype derived from the specification should be exercised with the users participating in the user requirements analysis process. The specification can then be reasoned with and expected behavior can be validated. The benefits of rapid prototyping have been identified to include [22][23]:-

- Rapid prototyping is available in the front end of the Software life cycle to allow early detection of errors.
- Unclear and imprecise user requirements can be clarified by rapid prototyping.
- Execution of the prototype supplements inspection and formal reasoning as means of analysis of the specification.
- The underlying theory of the TUG specification language is DCGs, which can be executed directly in the prolog environment .There is a close correspondence between TUG and prolog, which makes the process of transformation relatively mechanical. In this approach, DCGs are used as an intermediate form for aiding the transformation process. Although DCGs are syntactic for prolog, a prototype in DCGs seams difficult to read, understand, and maintain.
- Whenever there is a change in the user requirements, there many are no need to rederive the prototype from scratch if the change is trivial. ACRS is written to update the prototype only in response to the revised specification .A redervation of prototype in prolog from the start is avoided is modified [23] [24].
- The rapid prototyping approach supports formal requirement specifications written in TUG.
- The prototype is exercised to demonstrate the system behavior in the prolog environment. A driver that reads in the input data and then calls the main program with parameters needs to be constructed manually. The set of transformation rules are given below. The conventions are:
  a. C: is a finite set of condition tests and has the for
  b. {C1, C2,….,Cn} with n ≥ 1 where c_i is a TUG condition test;
  c. Y is a finite set of dummy non terminal or terminal node;
  d. Names of predicates in prolog are in all lowers case letters.
  e. Names of variables in prolog are in all upper case letters.
  f. Q is a finite set of prolog procedure calls and has the form

{q_1, q2…q $_n$} with n ≥ 1 where q_i is a prolog predicate for which a Prolog definition has been given, and. < > encloses optional syntactic Items

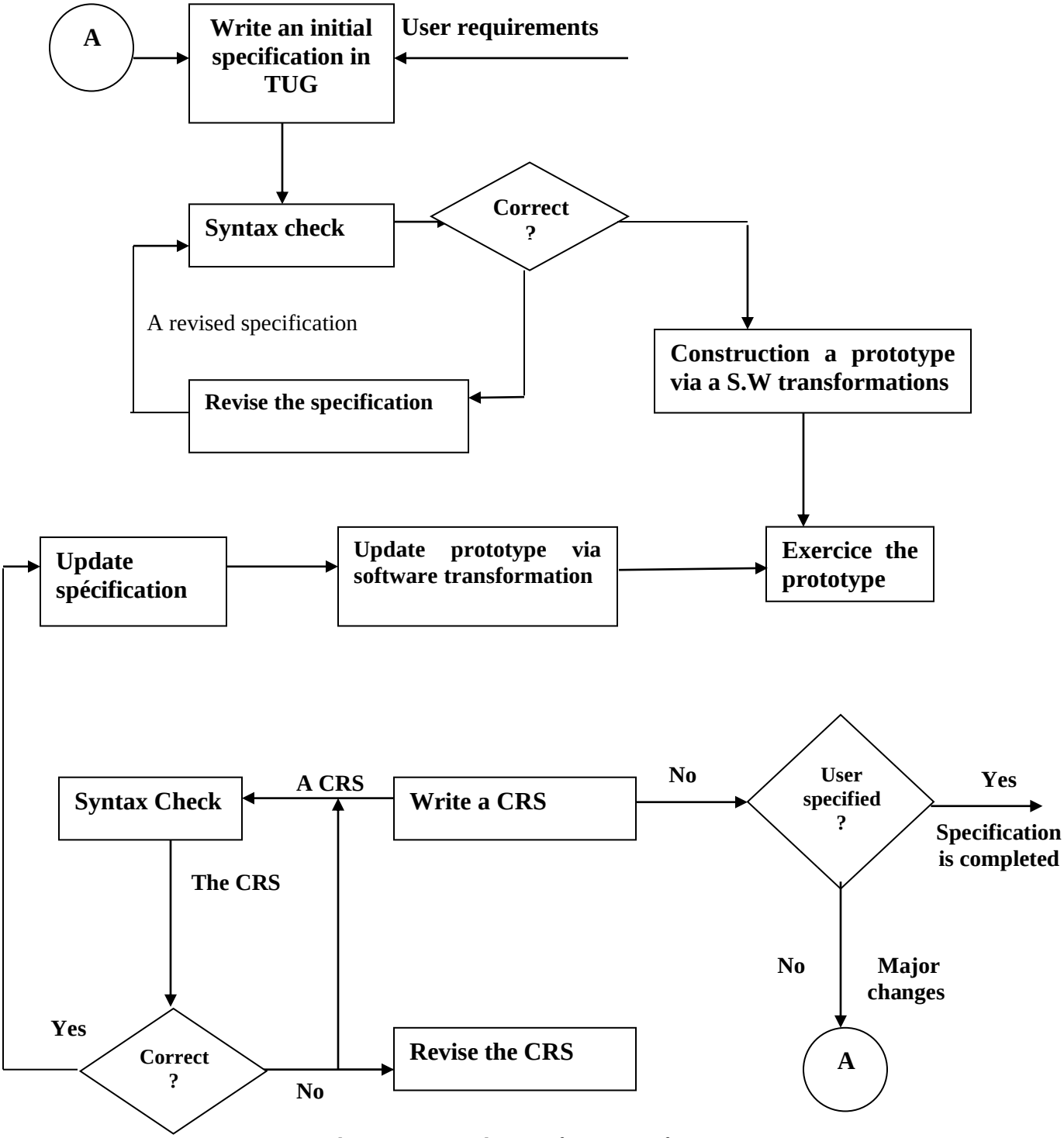

Fig .5 Rapid Prototyping Cycle via Software Transformations

The following four rules translate the analysis part of the TUG specification into DCGs. Each non-terminal node in the analysis tree structures its subtrees according to one of the structure notations each structuring operation can be transformed into a DCG form by applying the following four rules in straight forward manners[23][24][25].

|-def 1
α1
β1 where α is a nonterminal node
< {Φ1}>

β_2
$< \{ \Phi_2\}>$
……..
β_n
$<\{\Phi_n\}>$
$\alpha \rightarrow \beta_1 < \{ \Phi_1\}>$
$\alpha \rightarrow \beta_2 < \{ \Phi_2\}>$
……
$\alpha \rightarrow \beta_n < \{ \Phi_n\}>$

In rule1, a union nonterminal node α in the analysis tree indicates that α is one of the alternatives, β1, β2,….., β_n. If β_i is a literal, there is no Φ_i associated with β_i. Each translated DCG represents an alternative [9] [10].

9 lef 2
&
β_1 where α is a nonterminal node
$< \{\Phi 1\}>$ β_i is a nonterminal
or terminal node with condition tests Φ_i C
β_2
$<\{\Phi_2\}>$

β_n
$<\{ \Phi_n\}>$
$\alpha \rightarrow \beta_1 <\{ \Phi_1\}> \beta_2 <\{ \Phi_2\}>$ …….. $\beta_n <\{ \Phi_n\}>$

In rule 2, α a concatenation nonterminal node α in the analysis tree indicates that & is a concatenation of β_1, β_2,….., β_n. If β_i is a literal, there is no Φ_i associated with β_i. Each translated DCG represents a concatenation form

*def 3
α *
β_1 where α is a nonterminal node
$< \{\Phi_1\}>$ β_1 is a nonterminal
or terminal node with condition tests Φ_i C
β_2
$< \{ \Phi_2\}>$

β_n
$<\{\Phi_n\}>$
$\alpha \rightarrow [\ ]$
$\alpha \rightarrow \beta_1 <\{ \Phi_1\}> \beta_2 <\{ \Phi_2\}>$. ….. …….. $\beta_n <\{ \Phi_n\}> \alpha$

In rule 3, a kleene closure nonterminal node α in the analysis tree indicates that α is a sequence of zero or more occurrence of β_1, β_2,….., β_n. If βi is a literal, there is no Φ associated with β_i two translated DCGs represent a kleene closure form [5][11].

+ - def 4
α +
β_1 where α is nonterminal node
$<\{ \Phi_1\}>$ β_i anonterminal or terminal node with condition tests Φ_i C
β_2
$< \{ \Phi_2\}>$

β_n
$<\{\Phi_n\}>$
$\alpha \rightarrow \beta_1 <\{ \Phi_1\}> \beta_2 <\{ \Phi_2\}>$ …….. $\beta_n <\{ \Phi_n\}>$
$\alpha \rightarrow \beta_1 <\{ \Phi_1\}> \beta_2 <\{ \Phi_2\}>$ …….. $\beta_n <\{ \Phi_n\}> \alpha$

In rule 4, α a positive closure nonterminal node α in the analysis tree indicates that α is a sequence of one or more occurrences of β_1, β_2... β_n. If β_i is a literal, there is no Φ_i associated with β_i.. Two translated DCGs represent a positive closure form [25][26].To demonstrate the use of transformation rules presented in this section, the application of Rules 1-4 to the analysis tree of the related work approach specification produces the following results:

(1) SEQUENCE→UNSORTED- ID_S
(2) SEQUENCE →SORTED- ID_S
(3) UNSORTED→ List 1 +X{integer(X)}
List 2+Y {integer(y)},
Greater-than(X, Y)}
Rest_of_elements
(4) SORTED →ASCENDING-SEQUENCE

(5) ASCENDING_SEQUENCE→ Element {integer (element)}
(6) ASEDING_SEQENCE→ Element {integer (element)}
ASCENDING-SEQUENCE

The following four rules translate the anatomy tree of the specification into DCGs. The rules are similar to the rules for translating the analysis tree. The difference is that we use the**":-** "symbol instead of symbol "**→**". The use of the " **→**" symbol in the rules for the analysis part pf a TUG specification denotes a derivation of a tree, an involvement of pattern matching, and an unification of variables with the input values in prolog and the use of the**" :-**" symbol in the rules for the anatomy part of the specification perfumes the same operations. The uses of the **"→** "and"**:-**" symbols are just for the syntactic purpose. The out puts of the rules for the analysis part of a TUG specification produce a tree unified with the input values that is the input to the rules for the anatomy part of the specification. The rules for the anatomy part of a TUG specification reads in the tree and performs exact unifications on the variables to produce outputs Another difference is that dummy nodes appear in the rules. The reason for having dummy nodes is that often only the parts of tree are referenced in the anatomy tree of the specification. The remaining unreferenced parts of the tree still need to be unified in the course of patter matching. Dummy nodes are obtained by referring back to the analysis tree of the specification [17] [19].

| - def 5
α |
β_1 where α is a nonterminal node
β_2 β_i is a nonterminal node
……..
β_n
ε: -Y_1 where ε is uppercase using α
ε: -Y_2 where Y_i is uppercase using β_i
…….
ε: -Y_n

In Rule 5, a union nonterminal node α in the anatomy tree indicates that α is one of the alternatives, β_1, β_2,…, β_n .Each translated DCG represents an alternative.

&-def 6
α &
β_1 where α is a nonterminal node
β_2 , β_i is a nonterminal node or statement
……
β_n
ε:-Y_1<{Ψ_1}> Y_2<{Ψ_2}>…… Y_n<{ Ψ_n}> Where Ψ_1 C Y
ε is uppercase using α
Y_i is uppercase using β_i if β_i is a nontreminal node;
Otherwise $Y_i = \beta_i$

In Rule 6, a concatenation nonterminal node α in the anatomy tree indicates that α is a concatenation of β_1, β_2,…, β_n. The translated DCG represents a concatenation form.

* -def 7
α *
β_1 where α is a nonterminal node
β_2 β_i is a nonterminal node ,terminal node, or statement
…….
β_n
ε:- [ ]
ε:- Y_1<{Ψ_1}> Y_2{Ψ_2}>…….Y_n<{Ψ_n}> ε where Ψ_1 C Y
ε is uppercase using α
Y_i is uppercase using β_i if β_i is a nontreminal node;
Otherwise $Y_i = \beta_i$

In Rule7, a kleene closure nonterminal node α in the anatomy tree indicates that α is a sequence of zero or more occurrence of β_1, β_2,…, β_n. Two translated DCG$_s$ represent a kleene closure form.

+-def 8
α +
β_1 where α is a nonterminal node
β_2 β_i is a nonterminal node ,terminal node, or statement
……
β_n
ε:- Y_1<{Ψ_1}> Y_2{Ψ_2}>…….Y_n<{Ψ_n}>
ε:- Y_1<{Ψ_1}> Y_2{Ψ_2}>…….Y_n<{Ψ_n}> ε

Where Ψ_1 C Y

ε is uppercase using α

Y_i is uppercase using β_i if β_i is a nontreminal node;

Otherwise $Y_i = \beta_i$

In Rule 8, a positive closure nonterminal node α in the anatomy tree indicates that α_{is} a sequence of one or more occurrences of $\beta_1, \beta_2, \ldots, \beta_n$.

Two translated DCG represent a positive closure form. The application of Rules 5-8 to the anatomy tree of the problem specification produces the following results

(7) SEQUENCE:- UNSORTED
(8) SEQENCE:-SORTED
(9)UNSORTED: -T-L1=Y:: List 2
T-L2=X:: rest_of_elements
T-L=List 1< >T-L1< >T-L2
Call ID_S_sort (T-L)
(10) SORTED: - ASCENDING_SEQUENCE
(11) ASCENDING_SEQUENCE:-Output element
Output ' '
(12) ASCEDING_SEQENCE:-Output element
Output ' '
ASSCENDING_SEQUENCE

V. A TUG Specification for Implementing Atomic Read/Write Shared Memory in Mobile Ad Hoc Networks Application

This section will illustrates the usage of TUG for implementing atomic read/ write shared memory in mobile ad hoc networks. A specification in TUG is formalized incrementally in a modular and Top-down manner the example also illustrates how the language supports module independence via the language patterns. The Geoquorums approach, for implementing atomic read/ write shared memory in mobile ad hoc networks. This approach is based an associating abstract atomic object, with certain geographic locations. We assume the existence of local points, geographic areas that are normally "populated" by mobile nodes. The Geoquorum algorithm uses the fault –prone focal point objects to implement atomic read/write operations on fault – tolerant virtual shared object. Te Geoquorums algorithm uses a quorum- based strategy in which each quorum consists of a set of focal point objects. The quorums are used to maintain the consistency of the shared memory and to tolerate limited failures of the focal point objects which may be caused by depopulation of the corresponding geographic areas .Overall, the new geoquorums algorithm efficiently implements read and write operations in a highly dynamic, Mobil network.

A. A First Attempt at the Specification

```
MODULE a_ listing_ of _transitions
(in: TRANSITION_TYPE)
ANALYSIS
TRANSITION _TYPE &
TRANSITIONS *
"Put"
"get"
"confirm"
"config"
"reconfig"
END OF ANALYSIS"
ANATOMY
Transition_ type&
Transition*
Output n1
Output   "put"
Output "get"
Output "confirm"
Output "config"
Output "reconfig"
END OF ANATOMY
END OF MODULE a_ Listing_ of_ transitions
```

B. The Application of Transformation Rules to the Above Specification Module Results

In the Following Prototype in Prolog:

```
Transition_ type (transition_ type
(TRANSITION))→
```

```
Transition (TRANSITION).
Transition (Transition ([ ]))→
[ ].
Transition (transition ("put", "get", "confirm", "config", "reconfig")) →
["put"],
["get"],
["confirm'],
["config"],
["reconfig"]
Transition (TRANSITION).
Transition_ type (transition_ type
((TRANSITION)):-
Transition (REANSITION).
Transition (Transition ([ ])).
Transition (Transition("put", "get", "confirm", "config", "reconfig")) :-
n1,
Write ("put", "get", "confirm", "config", "reconfig")),
Transition (TRANSITION).
```

C. The Following CRS is Further Refinement on Each Invocation

```
Replace TRANSITION* under TRANSITION _Type &
With
VARIABLE_TYPE_TRANSITION.
"put_ invocation"
"get_ invocation"
"confirm_ invocation"
"config _invocation"
"reconfig_ invocation"
Replace Transition* under transition_ type &
With
Variable_Type_Transition |
Variable Transition &
Output n1
Output "put _ invocation"
Output" get _ invocation"
Output "confirm – invocation"
Output "config – invocation"
Output" reconfig – invocation"
```

D. A CRS for This Refinement is shown below

```
Replace PUT_INVOCATION_SECTIONS* under PUT_ INVOCATION
With
Put_ invocation
{new_value (put_invocation)
New_tag (put_ invocation)
New_config_id (put_ invocation)}
"get- invocation".
ReplaceGET_INVOCATION_SECTIONS*underGET_ INVOCATION & with
get_invocation
{new_config_id (get_invocation)}
"get_invocation"
```

At this stage, the application of transformation rules to the above two CRSs result in the following Prolog to update the prototype.

```
Transition_Type (transition_type (TRANSITION)) →
transition (TRANSITION).
transition (transition ([ ])) → [ ]
transition (transition(PUT_INVOCATION,GET_INVOCATION, CONFIRM_INVOCATION, CONFIG_ INVOCATION,
RECONFIG_INVOCATION)) →
put_invocation (PUT_INVOCATION),
(GET_INVOCATION).
transition (TRANSITION)
Variable_type_transition(put_invocation (PUT_INVOCATION)) →
```

put_invocation (PUT_INVOCATION).
Variable_type_transition(get_invocation (GET_INVOCATION)) →
get_invocation (GET_INVOCATION).
put_invocation('put_invocation_separator' PUT_INVOCATION_SECTION)) →
['put_invocation_separator'],
put_invocation_section (PUT_INVOCATION_SECTION).
put_invocation_section ([ ])).
put_invocation_section(put_invocation_section(PUT_INVOCATION,'put_invocation_separator',
PUT_INVOCATION_SECTION)) →
[PUT_INVOCATION],
{put_ack_response (PUT_INVOCATIO, NEW_VALUE, NEW_TAG,
NEW_CONFEG_ID),
Length (new_config_id >config_id)
[put_invocation→ PUT_ACK_RESPONSE],
Put_invocation_section (PUT_INVOCATION_SECTION).
Transition_type (transition_type (TRANSITION)): -
transition (TRANSITION)
transition (transition ([ ])).
Transition(transition(PUT_INVOCATION,GET_INVOCATION,CONFIG_INVOCATION,RECONFIG_DONE
_INVOCATION)): -
get_invocation (GET_INVOCATION),
transition (TRANSITION).
Get_invocation (get_invocation (GET_INVOCATION)):-
get_invocation (GET_INVOCATION).
confirm_invocation (confirm_ invocation (CONFIRM- INVOCATION)):
confirm_ invocation (CONFIRM_ INVOCATION).
get_invocation(get_invocation ('initial_get_invocation_separator',
GET_INVOCATION_SECTION)):-
write ('get_invocation'),
write ('put_invocation'),
write ('config_invocation'),
write ('reconfig _done_invocation'),
confirm_invocation(confirm_invocation('new_tag',
CONFIRM_INVOCATION_SECTION)): -
nl,
write ('put_invocation'),
write ('get_invocation'),
write ('config_invocation'),
write ('recon_done_invocation'),

D. The Further Refinement

Replace 'put_invocation' under
NEW_CONFIG_ID & with
PUT_ACK_RESPONSE
Stop &
'Stop'
' '
replace 'get_invocation' under
NEW_CONFIG_ID & with
GET_ACK_RESPONSE
Stop&
'Stop'
' '
replace 'config_invocation' under
NEW_TAG & with
CONFIG_ACK_RESPONSE
Stop&
'Stop'
' '
replace 'recon_done_invocation' under
NEW_CONFIG_ID & with

```
RECON_DONE _ ACK
Stop&
'Stop'
'  '
replace transition_ type & with
output 'transition Analysis'
output nl
transition*
output 'transition:'
output 'put_invocation'
output 'put _ ack _ response'
output 'get_invocation'
output 'get _ ack _ response'
output '.'
output nl
config_invocation &
output ' config_invocation'
output ' config _ ack _ response'
output ','
output ' recon_done _ invocation'
output ' recon_done _ ack'
output '.'
output n1
```

After a successive of refinements to the original specification, the final complete specification for implementing atomic read/write shared memory in mobile ad hoc networks is shown below.

```
MODULE a _ Listing _ of _ Transitions
(in: TRANSITION _ TYPE)
ANALYSIS
TRANSITION _ TYPE &
TRANSITION*
PUT_INVOCATION |
NEW_CONFIG_ID&
PUT_ACK_RESPONSE
Stop&
'Stop'
Get_ INVOCATION |
NEW_CONFIG_ID&
GET_ACK_RESPONSE
Stop&
'Stop'
'  '
CONFIG_INVOCATION |
NEW_TAG&
CONFIG_ACK_RESPONSE
Stop&
'Stop'
'  '
RECONFIG_DONE_ INVOCATION |
NEW_CONFIG_ID&
RECON_DONE_ACK
Stop&
'Stop'
'  '
END OF ANALYSIS
transition type &
output' Transition Analysis'
output nl
transition*
output' transition:'
put _invocation |
```

```
new_config_id &
output 'put_ invocation'
output 'put_ack_response'
output '. '
output nl
get _ invocation |
new_config_id &
output 'get_ invocation'
output 'get_ack_response'
output ', '
config- invocation |
new-tag &
output 'config- invocation'
output 'config -ack-response'
output ' , '
recon - done- invocation |
new-config –id &
output ' reconfig - done-  invocation'
output ' recon - done- ack'
output '.'
output nl
END OF ANATOMY;
END OF MOUDULE a- Listing-Of- Transitions.
```

VI. Conclusions and Future Work

An approach was developed to support rapid prototyping via software transformations by deriving a prototype in prolog from a specification in TUG. i didn't directly use the prolog language to specify user requirements, programming languages more or less concentrate on how rather than what and are generally unsuitable for specification purposes. In addition, in a specification in prolog lacks modularity in contrast to a specification in TUG. Since the main purpose of a specification is to aid the understanding of the user requirements, it is useful if a specification can be read and understood. Modularity helps specifiers to read and maintain in a manageable way. TUG provides modularity to help specifiers to specify a system in a hierarchical manner. A set of modules are specified and then composed into a system. The system is tested in pieces corresponding to the modular specification .In contrast to a specification in TUG, a specification in prolog is relatively difficult to maintain. Rapid prototyping via software transformations helps to build prototypes automatically from specifications. In this paper, a formal method with TUG was presented to support the rapid prototyping via software transformations process in which a prototype can be built quickly and cheaply. Automation of the application of software transformations reduces the labor intensity of developing prototypes manually. Rapid prototyping via software transformations also provides support for prototype modifications. The rapid prototyping approach supports prototype evolution by avoiding complete retransformation of the prototype from the start whenever there is a change made to the specification. To avoid complete retransformation, a CRS is written to update the prototype only in response to the minor changes to the specification, involving the nodes to be modified, extended, relaxed, or refined. If a major change is needed, the specification may need to be rewritten and a new prototype may be derived from the start. A major change may involve the structure of the specification to be modified. Like other formal specification languages, the TUG specification language may not provide enough abstractions for modeling some properties of software systems such as non- functional properties. Therefore, the geoquorum approach for implementing atomic read/write shared memory in mobile ad hoc networks encourages specifiers to manually add additional code to the derived prototype for demonstrating such kind of properties of systems. The rapid prototype approach supports the quick construction of a prototype with a high degree of module independence. Module independence has a particular importance in this approach because of the need for modifications to the prototype. It remains an open question to determine how to choose a good set of focal points, how to construct a map of focal points in a distributed fashion, and how to moodily the set of focal points dynamically. Overall, the FPO Model will significantly simplify the development of algorithms for mobile, in highly dynamic networks. Finally, there exist many techniques to do these phases of software development lifecycle for any application.

References

1. J. Weng, C. Miao, A. Goh, "Protecting Online Rating Systems from Unfair Ratings," International Conference on Trust, Privacy and Securityin Digital Business, pp. 50–59, 2005.
2. S.I. Ahamed, M.M. Haque, M.E. Hoque, F. Rahman, N. Talukder, "Design, analysis, and deployment of formal trust model (FTM) with trust bootstrapping for pervasive environments," Journal of Systems and Software, vol. 83, No. 2, p. 253–270, 2010.
3. N. Iltaf, A. Ghafoor, U. Zia, " A mechanism for detecting dishonest recommendation in indirect trust computation," EURASIP Journal on Wireless Communications and Networking, vol. 189, 2016.

4. Barmade , M.M. Nashipudinath, "An efficient strategy to detect outlier transactions," International Journal of Soft Computing and Engineering (IJSCE), vol. 6, no. 174-178, p. 3, 2014.
5. Z. He, X. Xu, J.Z. Huang, S. Deng, "Fp-outlier: frequent pattern based outlier detection," Computer Science and Information Systems, vol. 2, no. 1, pp. 103-118, 2015.
6. F. Hendrikx, K. Bubendorfer, R. Chard, "Reputation systems: a survey and taxonomy," Journal of Parallel and Distributed Computing, vol. 75, pp. 184-197, 2015.
7. Dobson, A. J., and A. G. Barnett. , An Introduction to Generalized Linear Models, Chapman and Hall/CRC. ,Taylor & Francis Group, 2018.
8. A . Manna, A . Sengupta, C. Mazumdar, "A survey of trust models for enterprise information systems," Procedia Comput. Sci., vol. 85, p. 527– 534, 2016.
9. R. Malaga, "Web-based reputation management systems: problems and suggested solutions," Electronic Commerce Research, p. 403–417, 2017.
10. G. D'Angelo, S. Rampone, F. Palmieri, "An artificial intelligence-based trust model for pervasive computing," Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), p. 701–706, 2015.
11. G. D'Angelo, S. Rampone, F. Palmieri, "Developing a trust model for pervasive computing based on apriori association rules learning and Bayesian classification," Soft Comput., p. 6297–6315, 2017.
12. Gianni D'Angelo , Francesco Palmieri , Salvatore Rampone, "Detecting unfair recommendations in trust based pervasive environments," Information Sciences, -Vol 17,no.8,pp. 113-126,2019
13. J. Weng, C. Miao, A. Goh, "Protecting Online Rating Systems from Unfair Ratings," International Conference on Trust, Privacy and Securityin Digital Business, p. 50–59, 2005.
14. C.Chiang,J.E.Urban, "Validating Software Specification against User Claims", Proceedings of the Twenty-Third Annual International Computer Software and Applications Conference _OMPSAC 2010),2010,PP:104-109.
15. DOLEV, S., Gilbert, S.LYNCH, N.A., SHVARTSMAN, A.A., Welch, J.L.: " Geoquorums: Implementing Atomic Memory in Mobile Ad Hoc Networks". In: Proceeding of the 17th International Conference on Distributed Computing, PP. 306-320 (2018).
16. Haas, Z.J., Liang, B.: "Ad Hoc Mobile Management with Uniform Quorum Systems". IEEE/ACM Transactions on Networking 7(2), PP: 228-240 (2000).
17. B.CMoszkowski,"A Complete Axiomatization Of Interval Temporal Logic With Infinite Time ", Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science_(LICS·00),JUNE(2000),26-29,California ,2017, PP:242-249.
18. Chia-Chu Chiang," Automated Rapid Prototyping of TUG Specifications Using Prolog", Proceedings of: Information and Software Technology 46(2014), PP: 857-873.
19. O.J.Dahl, O. Owe, Formal Methods and the RMODP, Research Report No.261, Department of Information, University of Oslo, Norway, 2000.
20. IEEE, IEEE standard for a High Performance Serial Bus, Standard 1394, August 1995.
21. M.Liu Yanguo, Proof Patterns for UMI-Based Verification, Master Thesis ECE Department, University of Victoria, Victoria, Canada, October 2002.
22. I.Traore, D.Aredo, H.Ye," An Integrated Framework for Formal Development of Open Distributed Systems," In: Information and Software Technology 46 (2004) 281-286.
23. El-Far, Automated Construction of Software Behavior Models, Master's Thesis, Florida Institute of Technology Melbourne, Fl, 1999.
24. G. Carullo, A. Castiglione, A. De Santis, F. Palmieri, "A triadic closure and homophily-based recommendation system for online social networks," World Wide Web, vol. 18, no. 6, pp. 1579–1601, 2015.
25. Tharwat, "Classification assessment methods," Applied Computing and Informatics, vol. 17, no. 1, pp. 168-192, 2019.