

# Formal Analysis of Tree-Based Aggregation Protocols

RENA BAKHSI



**KTH Information and  
Communication Technology**

Master of Science Thesis  
Stockholm, Sweden 2006

IMIT/LECS-2006-08

# Formal Analysis of Tree-Based Aggregation Protocols

RENA BAKHSI

Master of Science Thesis  
Stockholm, Sweden 2006

IMIT/LECS-2006-08

## **Abstract**

In distributed systems, information aggregation is a problem of summarizing data collected from multiple sources, e.g. nodes, and combined using an aggregation function. The task of a tree based aggregation protocols is to compute an aggregation function in a more efficient, decentralized way using an aggregation tree. We study the problem of formal correctness analysis of such protocols. Our formal analysis consists of two parts. The first part is the clarification of common concepts for the class of tree-based aggregation protocols including design requirements and properties. The second part is case study of Generic Aggregation Protocol (GAP). In the second part, the GAP protocol is examined for several requirements, including deadlock freedom, liveness and safety properties using UPPAAL – a model checking tool for networks of timed automata.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background	1
1.1.1	Communication	2
1.1.2	Timing models	2
1.1.3	Stabilization	2
1.1.4	Network	2
1.1.5	Network management	3
1.1.6	Aggregation	4
1.1.7	Formal analysis	5
1.2	Goal of the thesis	5
1.3	Related Work	6
<b>2</b>	<b>Tree-Based Aggregation Protocols</b>	<b>7</b>
2.1	Policies	8
2.2	Related Protocols	9
2.3	GAP Protocol	9
2.3.1	Protocol Description	9
2.3.2	The algorithm	11
<b>3</b>	<b>Case Study with UPPAAL</b>	<b>14</b>
3.1	Background	14
3.1.1	Timed Automata	15
3.1.2	UPPAAL	16
3.2	Modelling the GAP Protocol	23
3.2.1	Architecture of the model	23
3.2.2	Timing Information	24
3.2.3	Node template	25
3.2.4	Discovery template	27
3.2.5	Buffer template	28
3.3	Verification	29
3.3.1	Configurations	29
3.3.2	Validation	31
3.3.3	Requirements	35
3.4	Discussion	37
<b>4</b>	<b>Conclusion</b>	<b>39</b>

---

<b>Bibliography</b>	<b>41</b>
<b>A Pseudocode</b>	<b>44</b>
A.1 GAP Pseudocode . . . . .	44
A.2 TCA-GAP Pseudocode . . . . .	46
<b>B GAP code in UPPAAL</b>	<b>48</b>
B.1 Full model . . . . .	48
B.1.1 System global code . . . . .	48
B.1.2 Node template . . . . .	49
B.1.3 Buffer template . . . . .	52
B.1.4 System declaration . . . . .	53
B.2 Simplified model . . . . .	53
B.2.1 System global code . . . . .	53
B.2.2 Node template . . . . .	54
B.2.3 Buffer template . . . . .	58
B.2.4 System declaration . . . . .	58

# Figures

1.1	Distributed system example	1
1.2	Computer network example (with server)	3
1.3	Tree graph example	3
1.4	Network monitoring example	4
2.1	Aggregation tree network example	7
2.2	Example of BFS tree	10
2.3	Main loop of the algorithm	12
2.4	The procedure <code>retoreTableInvariant()</code>	13
3.1	The Vending machine example	15
3.2	Synchronous value passing example in UPPAAL	17
3.3	Syntax of expressions in BNF	17
3.4	Syntax of types in BNF	18
3.5	Syntax of functions in BNF	18
3.6	GAP's Node template	19
3.7	Variable reduction	20
3.8	Reachability property example	21
3.9	Safety property examples	21
3.10	Liveness property examples	22
3.11	Architecture of a node in the GAP model in UPPAAL	24
3.12	Model for three nodes	24
3.13	Node template	25
3.14	Discovery template	27
3.15	Buffer template	28
3.16	Tree graph with three nodes (Config. 1)	29
3.17	Tree graph with three nodes (Config. 2)	30
3.18	Tree graph with three nodes (Config. 3)	30
3.19	Tree graph with four nodes (Config. 4)	30
3.20	Tree graph with four nodes (Config. 5)	31
3.21	Simplified Node template	35
3.22	Simplified Buffer template	35

# Tables

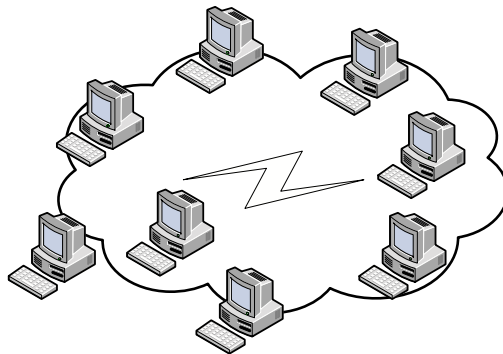
2.1	Sample neighbourhood table of arbitrary node . . . . .	11
2.2	Initial neighbourhood table of the root node . . . . .	11
3.1	Correspondence between CTL and UPPAAL query language syntax . . . . .	20
3.2	Neighbourhood table's and vector arrays' "Index-Field" correspondence . . . . .	26
3.3	Global shared variables . . . . .	26
3.4	BFS construction bfsT() . . . . .	34
3.5	Deadlock verification . . . . .	36
3.6	Liveness verification . . . . .	37
3.7	Safety verification . . . . .	37

# Chapter 1

## Introduction

### 1.1 Background

A *distributed system* (Fig. 1.1) is an interconnected collection of autonomous computers, processors, processes, or simply *nodes* [34]. The term *autonomous* is used in the sense that a node's resources are managed locally. Nodes are *interconnected* if they are able to exchange the information. In distributed computing theory, the term *model* is usually used to denote abstract representation of a distributed system.



*Figure 1.1: Distributed system example*

An *algorithm* is a program given to the processors to solve a particular task on a defined model setting. In order to compare different algorithms for the same problem it is useful to measure the consumption of resources by an algorithm. *Complexity analysis* of an algorithm can be done in terms of complexity parameters such as bandwidth consumed (*bit complexity*), number of exchanged messages (*message complexity*), time consumed by a computation (*time complexity*) and amount of memory needed in a process to execute the algorithm (*space complexity*). A *protocol* is a distributed algorithm executed by several parties in order to achieve a common goal. Agents participating in a protocol are called *units* (or *entities*). Typically, a unit receives a message from another unit, examines it, and executes further actions such as composing and sending other messages as well as internal instructions. The exact sequence of actions performed by an unit is determined by its role in a given run of the protocol. Units can engage in multiple protocol runs and even take on multiple roles simultaneously.

There are various models of a distributed system with some alternative choice for several components.

### 1.1.1 Communication

The communication model of a distributed system describes the information exchange mechanism between processors. Different models are distinguished by the mechanism employed for interprocess communication – *message passing* or *shared memory*.

In the message passing model, processes communicate by sending and receiving messages – a process sends a message by adding it to its outgoing message queue, and receives a message by removing it from its incoming queues. Queues may be of bounded or unbounded size. In the first case, the model should specify the situation when a message is added to a full queue. Usually, either the last arrived message is lost or this message can replace the other one in queue, which means that the other message is lost. Furthermore, each processor may have one queue for all incoming messages or a separate queue for each adjacent communication channel. Also, the communication model can include the ability to send a message to a specific neighbour in one step, or send different type of messages to each neighbour in one step, or broadcast/multicast a single message to its neighbours.

In shared memory model, processors communicate through globally shared objects, e.g. registers.

### 1.1.2 Timing models

There are two timing models in distributed systems – *synchronous* and *asynchronous*. In the synchronous model, there is a mechanism by which each process' action is synchronized with a global clock. In the asynchronous model, processes execute their algorithms at different speeds and there is no strict assumption about how long the execution may take. It is possible to achieve some degree of synchrony in the system by using, for instance, synchronizers [34].

### 1.1.3 Stabilization

*Stabilization* is one of the most important requirements to provide fault-tolerance in distributed systems. One type of stabilization is *robustness*, in which the protocol follows a pessimistic approach. It assumes that initial configuration is safe and ensures the correct operation of the protocol by constantly "suspecting" failures in the system and running a specific algorithm. Another type is *self-stabilization*, in which the protocol follows an optimistic approach in the sense that the distributed system may be affected by failures, but guaranteed to recover from any arbitrary state in finite time.

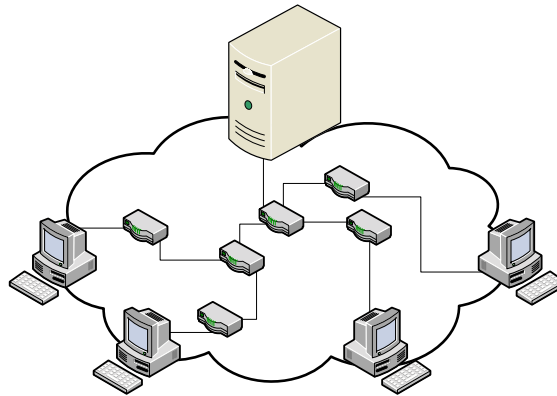
### 1.1.4 Network

Computer networks is an example of a distributed system. A *computer network* is a collection of computers, connected by communication mechanisms by means of which the computers can exchange information (Fig. 1.2), e.g. by message-passing.

A network is *semi-uniform* if there is one process (the root) which executes a different algorithm. Thus, a network is *uniform* if all processes execute the same code.

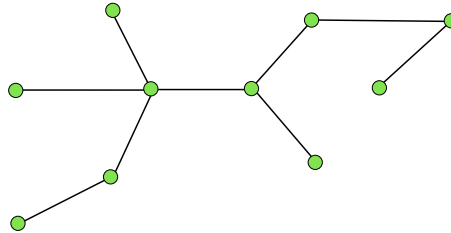
In general, a message-passing system is modelled as a *graph*  $G = (V, E)$ , where processors are the nodes and the communication links are edges between two nodes, if they are neighbours. A *network topology* is the pattern in which nodes of a network are connected by links and communicate with each other. An example of network topology includes ring, tree, chain and etc.

A *tree* on  $n$  nodes is a connected graph with  $n - 1$  edges (Fig. 1.3); it contains no cycles. Trees are used in distributed computations because they allow computation at a low communication cost, and, moreover, each connected graph contains a tree as a spanning subnetwork [34]. A *spanning tree* of a



*Figure 1.2: Computer network example (with server)*

connected, undirected graph is a tree composed of all vertices (nodes) and some edges of that graph; that is, no vertex is not connected to the tree.



*Figure 1.3: Tree graph example*

Breadth-first search (BFS) is a tree search algorithm used for traversing a tree, tree structure, or graph. The algorithm begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal. BFS tree results from a breadth-first traversal of the underlying network topology. This tree defines a shortest path from the root to every other node in the tree. And similar, depth-first search (DFS) trees are obtained from depth-first search traversal [23].

Network topology can be *dynamic* and *static*. In dynamic topology network, channels as well as processes can be added or removed from the system during the computation. On the contrary, a static topology remains fixed during the computation.

Some models assume that each process knows the complete set of processes, other models assume that a process stores information only about some processes, e.g. its immediate neighbours.

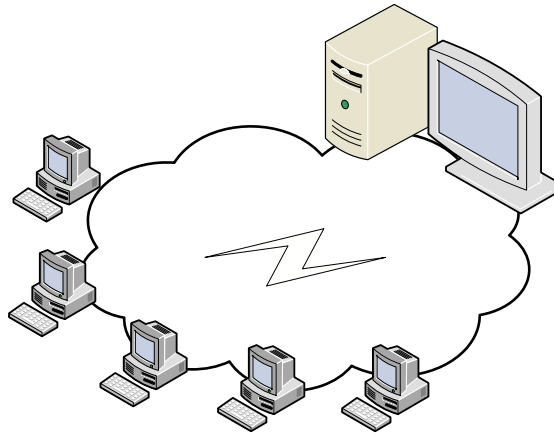
A *routing algorithm* is a decision-making procedure by which a node sends a message on its way to the selected subset of its neighbours. The optimality of a routing algorithm depends on the notion of "best path" – *minimum hop*, *shortest path* and *minimum delay* [34].

Our focus is on large and highly dynamic networks where nodes can join and leave the network at any time, links among nodes can be created and deleted based on the requirements of the particular application. Control and monitoring such distributed systems is a difficult task.

### 1.1.5 Network management

The process of controlling a network to maximise its efficiency and productivity refers as a *network management* [25]. Typically, it is applied to large scale networks such as telecommunication

networks. Network management includes the execution of the set of functions required for controlling, planning, allocating, deploying, coordinating, and monitoring the resources of a network. A *network monitoring* describes the use of a system that constantly monitors a computer network for slow or failing systems and that notifies the network administrator (or root) in case of outages via communication medium (Fig. 1.4).



*Figure 1.4: Network monitoring example*

Another notion in the network management area, an *aggregation* [36] refers to a set of functions that provide global information or summaries about configuration of the distributed system.

### 1.1.6 Aggregation

Assume a distributed system with  $n$  nodes, where  $n$  is very large number, and each node  $i$  has an information about its local state  $w_i(t)$  and the links to its neighbours. In the system, nodes can arrive, depart and fail over time or communications between nodes may be lost. The task is to compute a function  $f(w(t))$ , where  $f(\cdot) = \{f_1(\cdot), f_2(\cdot), \dots, f_n(\cdot)\}$  and  $w(t) = \{w_1(t), w_2(t), \dots, w_i(t)\}$ . The former means that each node in the network computes its knowledge on function  $f$ . As an example of a data aggregation, one might consider a system global state computation problem.

Keshav [22] divided the existing approaches to data aggregation into the following categories:

1. *Centralized*: In a centralized approach, a designated root node collects state information from all other nodes and computes an aggregation function. This approach is not scalable, because the root node becomes a bottleneck.
2. *Tree-based*: A tree-based approach is the generalization of the centralized approach. It consists of inducing a multi-level hierarchy or tree on the underlying graph. Each node sends its state information to its parent, which performs local aggregation and, in turn, sends the aggregated results upwards, eventually reaching the root. Like in a centralized solution, the loss of a single node can disrupt the tree.
3. *Flooding and Randomized flooding*: With flooding, a node, called infective node, which state has been changed, pushes its data to all or a random subset of its neighbours. They become infective, and forward this message to some or all their uninfected neighbours and so on.
4. *Random walk-based*: In a random walk approach, a node sends a message with its state information to a randomly selected neighbour. The latter uses this information to update its local

state, adds its local value to the message's state, and forwards the message to the next random neighbour. This is called *random walk*. More than one random walk may be in progress concurrently.

5. *Randomized gossip based*: In each round of computation of random gossip, every node talks to some randomly selected neighbours and exchanges some information with them.

A classification of aggregation functions according to other various dimensions is possible [27].

Aggregation protocols execute procedures for collecting and computation of an aggregation function from all nodes in a network. Basically, aggregation protocols can be divided in two categories: *reactive* and *proactive*. Reactive protocols respond to specific queries issued by nodes in the network. The answers are returned to the issuer of the query. Proactive protocols continuously provide the value of some aggregate to some or all nodes in the network, trying to follow reasonably quickly changes in the network topology or in the value being aggregated [29].

### 1.1.7 Formal analysis

Formal methods are a collection of notations and techniques for description and analysis of systems. The term *formal* is used in the sense that they are based on some mathematical theories, such as logic, automata or graph theory [32].

*Formal analysis techniques* can be used to ensure that a system meets its design specifications and operates to the satisfaction of its users, i.e. the correctness of a system, or seek for cases where it fails to do so.

*Assertional proof method* reduces the problem of reasoning about concurrent systems to that of reasoning about each individual action and focuses on the states of the system. An *invariant* is a correctness assertion that need to hold throughout the execution at the main control points.

*Automatic verification techniques* can be applied to *finite state systems*, including communication protocols. *Model checking* is an automated technique for verifying a desired behavioural property of a system using its model.

*Protocol verification* is an area of research which focuses on the problem of ensuring the logical consistency of the protocol specification, independent of any particular implementation, as well as the correctness and completeness. *Correctness* is concerned with functional correctness; that is the protocol interactions satisfy the service specifications and the required service will eventually be delivered. *Completeness* refers to taking into consideration of all the possible events and the examination of all options and services. *Consistency* of a protocol specification is a conformity between the descriptions of its various parts and adjacent layers.

Protocol verification requires a description of protocol properties which can be categorized into general and specific properties. General properties are common to all protocols that maybe be considered to form an implicit part of all service specification, e.g. the absence of deadlock and the arrival in some system state or set of states. The provision for all possible inputs is another general property. Specific properties of the protocol require specification of the particular service to be provided, e.g. reliable data transfer in a transport protocol.

## 1.2 Goal of the thesis

The work described in this thesis is a case study on tree-based aggregation protocols. It continues the previous studies on this class of protocols, but towards automatic verification.

Tree-based aggregation protocols possess specific challenges to modelling and verification. Our analysis proceeds through several steps. In the first part, an overview of tree-based aggregation protocols is given, including key metrics, related protocols, optimization policies (collected from existing resources) and the GAP protocol description. The next step is to create an UPPAAL model for the GAP protocol as a case study and verify the model. This also includes formalizing properties representing protocol requirements in the model.

### 1.3 Related Work

Dolev *et al.* [12] presented a self-stabilizing BFS spanning-tree construction algorithm for semi-uniform systems with a central daemon under read/write atomicity. In this algorithm, every node maintains two variables – a pointer to one of its incoming edges (this information is kept in a bit associated with each communication register), and an integer measuring the distance in hops to the root of the tree. The distinguished node in the network acts as the root.

All previous works on GAP protocol have been evaluated and analysed based on the experimental results. The work by Wuhib [37] has been done on the previous version of GAP protocol, with no rate limitation scheme. The work describes various scenarios and discusses results of the tests carried on different network topologies that show the performance of GAP during initialization, weight change, topology changes and a combination of the above.

The work by PannoZZo [31] presents the implementation of a new version of GAP and the comparison of performances between a tree based aggregation protocol, where a single node collects information from all nodes in the network and performs the aggregation, and a gossip-based aggregation protocol, where each node exchanges information with its neighbours and returns an estimation of the aggregation function.

Hefti [17] has implemented a simple overlay platform for decentralized monitoring protocols like Echo, GAP and TCA-GAP. He has adapted Delaunay Triangulation Protocol [26] as underlying neighbour discovery service for construction of an overlay network. This construction prevents the cases of partitioning of a network graph and nodes disconnection. The failure discovery and network graph's management has not been considered in the work.

All protocols related to the class of tree-based aggregations are described in the next chapter.

# Chapter 2

## Tree-Based Aggregation Protocols

This chapter focuses on a tree-based solution to data aggregation, briefly described in Sect. 1.1.6.

A protocol that uses tree-based aggregation approach for data aggregation constructs and maintains a spanning tree on top of the underlying network overlay. A *unit* of the tree, e.g. node, aggregates required information received from its children in its subtree and sends aggregation to its parent. All aggregated values will eventually reach the top or the *root* which is responsible for network management and information processing (Fig. 2.1). However, the routing scheme varies for different protocols, but the general strategy is the same as described above. In the presence of failure, a spanning tree can be rebuilt.

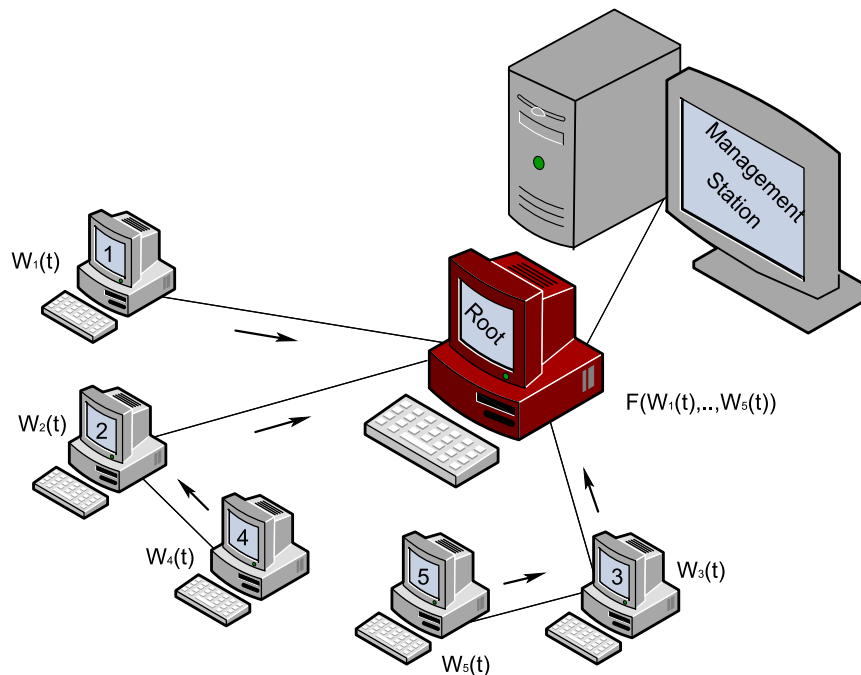


Figure 2.1: Aggregation tree network example

The communication cost of this solution is lower than for centralized solution. Besides, scalability of this approach is higher than of the centralized one [22, 31, 37]. Nevertheless, protocols fail to handle the situation with failure of the root.

The following key metrics should be addressed for the tree-based aggregation protocols:

- **Accuracy:** How close is a aggregated value to a real value of  $f(w(t))$ .
- **Efficiency:** Routing should proceed in finding paths with minimum hop count. Another aspect of efficiency is a complexity that includes the following: *message complexity*, the total number of messages exchanged by units of a system, executing the protocol; *bit complexity*, i.e. the amount of bytes exchanged by units to compute the aggregation function; *space complexity*, that is required to store a protocol's data; *time complexity*, which corresponds to the CPU time required to process tasks at each unit. The smaller the complexity to construct and maintain the network graph as well as to compute the aggregation function, the better is the protocol compared to other aggregation protocols.
- **Scalability:** Protocol should provide a support for large size of the network without significant loss in performance of the nodes.
- **Robustness:** Units of system (nodes) can depart or network links can fail at any time cause the disconnection of the part of network graph. Robustness specifies how sensitive is the computation of the global state in case of a topological change as well as to message loss. This determines the error in the computed function  $f(w(t))$  as a function of the fraction of nodes failed or messages lost.
- **Maintenance:** All protocols maintain a tree on top of the underlying network overlay that is used to collect data and compute the aggregation function. The approach taken to traverse a tree varies among the protocols, e.g. sink tree, BFS tree, DFS tree.
- **Convergence time:** That is, how much time it takes to reflect a local weight change on a node on the value of the aggregation function of the root node.

## 2.1 Policies

This section is an adaptation of [9, 37] since an additional simulation is needed to show the efficiency of described policies. Further information can be found in the source to see the results and observations.

There are several policies to optimize protocol behaviour with respect to the performance improvement, accuracy of the aggregations, convergence time and tolerance to failures (loss). These optimization techniques are applicable for all tree-based aggregation protocols. Some of these techniques are aggregation function dependent, i.e. they can be used only for particular classes of aggregates. There exists protocol-specific policies which will not be covered in this thesis, since we are interested in general concepts of this class of protocols as well as case study of particular protocols.

- **Conservative:** Once the parenthood relation changes, all information stored in the neighbourhood table except neighbour identities and levels and status of parent and self becomes, in principle, unreliable and could be forgotten (assigned the undefined value). This policy will ensure that the aggregate is, in some reasonable sense, always a lower approximation of the "real" value.
- **Cache-like:** All information stored in the neighbourhood table, except information concerning self or status of parent, is left unchanged. This appears to be a natural default policy, as it seems to provide a good compromise between overshoot/undershoot and convergence time.

- **Greedy or adaptive policies:** Other policies can be envisaged such as policies which attempt to predict changes in neighbour status, such as proactively changing the status of a peer once its level has seen to be 2 or more greater than self's. It is also possible to adapt the tree topology to MAC layer information such as link quality.

More details about experimental results on certain policies can be found in TAG [27] and works done on GAP and TCA-GAP [17, 31, 37].

## 2.2 Related Protocols

We now, briefly, consider several examples of protocols belong to this class.

*Border Gateway Reservation Protocol* (BGRP) [30] is a protocol for inter-domain resource reservation, in which bandwidth reservations are aggregated along sink trees from all data sources in the network. It scales well, in terms of message processing load, state storage and bandwidth. BGRP tolerates the route changes and repairs the routing tree using, so-called, *self-healing* process.

*Generic Aggregation Protocol* (GAP) [9] is a self-stabilizing protocol that compute an aggregation functions for network management purpose in distributed, scalable and robust manner. GAP builds the BFS tree on a network overlay, following the idea of Dolev *et al.* [11, 15]

*TCA-GAP Protocol* [38] is based on the GAP protocol, with respect to maintenance of the network graph and aggregating local variables, and aims to compute network threshold crossing alerts (NTCAs) in a scalable and robust manner.

*Tiny AGgregation Protocol* (TAG) [27] offers as a service for aggregation in low-power, distributed, wireless environments. TAG builds a routing tree using Ad-Hoc routing algorithm and tolerates disconnections and loss in robust manner. The advantage of this protocol is reducing of communication bandwidth.

*A-GAP Protocol* [33] is a protocol for continuous monitoring of network state variables with configurable accuracy. Like TCA-GAP, the protocol is based on the GAP protocol to manage the underlying network graph. In A-GAP, the accuracy is expressed in terms of the average absolute error, and is controlled by dynamically configuring filters in the management nodes.

This thesis is about a case study of tree-based aggregation protocol, GAP.

## 2.3 GAP Protocol

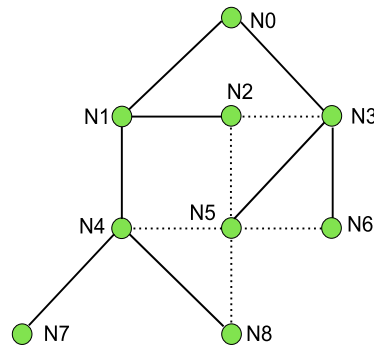
This section outlines the GAP protocol. It includes the protocol description, key design requirements for the protocol and an execution model of the protocol.

### 2.3.1 Protocol Description

Generic Aggregation Protocol (GAP) adapts a distributed scheme for computing a global state information or the aggregation function  $f(w_1(t), \dots, w_n(t))$  from local values  $w_i(t)$  received from node  $i$  at time  $t$ . The aggregation function  $f$  has the following properties: commutativity and associativity. The function also possesses an identity element 1. A local value (state variable) on a network node  $w_i(t)$  is referred to as generic term *weight*. An example of a weight is router load.

GAP is based on the algorithm that builds and maintains a BFS tree (Fig. 2.2) on top of a network overlay or network graph. This algorithm is inspired by the approach of Dolev *et al.* [11, 15]. A *root*

*node* of the tree is the start node of the protocol, which receives the propagated back aggregates from other nodes in the tree.



*Figure 2.2: Example of BFS tree*

In the GAP protocol, a distributed system is modelled as a set of processes linked together through a communication network. Processes communicate by message passing and each process reacts upon receipt of a message. That is, the protocol is an *event-driven model*. Thus, after the completion of an initial phase of the stabilization called the *initial stabilization*, when a BFS tree is constructed, nodes interact with each other only if a change in their states has occurred.

The communication network is assumed to be:

- *asynchronous*: the time taken by the communication network to forward a message to its destination can be arbitrarily long.
- *reliable*: messages are neither lost nor duplicated. Links might fail, but nodes don't.

The execution model of the protocol assumes a set of underlying services such as neighbour discovery and failure detection, local weight subscription, message delivery and timeout.

The key design requirements of the protocol can be summarized as follows:

- *Convergence time*: The time difference between a weight change on a node and this change has an effect on the aggregate on the root node, should be optimal.
- *Robustness*: When node departs, its weight will be removed from the aggregate until the node recovers. When new node arrives, its weight will be combined in the aggregate, all within tolerable delays and error margins.
- *Efficiency and scalability*: The load on the nodes must be balanced and the number of messages exchanged in the overlay must be small compared to a centralized aggregation protocol.

In order to calculate the aggregates correctly, each node holds and maintains information about itself, its children and parent in the BFS tree as its local neighbourhood table. Table 2.1 shows an example of such table.

In particular, the table contains an entry for each neighbour including its identity (ID), its status with respect to the current node, its level in the BFS tree, or distance to root in number of hops, as a non-negative integer, and its aggregate weight.

The status field can be one of the following: *self*, *parent*, *child* or *peer*. The value *self* defines a node itself, *parent* – node's parent, *child* – children of the current node and *peer* – a node which

Node ID	Status	Level	Weight
$n_1$	<i>child</i>	4	312
$n_2$	<i>self</i>	3	411
$n_3$	<i>parent</i>	2	7955
$n_4$	<i>child</i>	4	33
$n_5$	<i>peer</i>	4	567

**Table 2.1:** Sample neighbourhood table of arbitrary node

either has not been assigned any particular status or is not a neighbour in the aggregation tree. The aggregate weight is the aggregated weight  $f(w_1(t), \dots, w_n(t))$  of the spanning tree rooted in that node. The exception is *self*, where the weight field will contain only the weight of the local node.

Node ID	Status	Level	Weight
$n_i$	<i>self</i>	0	$w_i$
$n_{root}$	<i>parent</i>	-1	$w_{root}$

**Table 2.2:** Initial neighbourhood table of the root node

Initially, the neighbourhood table of all nodes  $n$ , except the root (see Table 2.2), contains a single entry  $(n, self, l_0, w_0)$  where  $l_0$  and  $w_0$  is some initial level and weight, respectively. The initial level must be a non-negative integer. In GAP, the network is uniform with respect to the stabilization, i.e. the same code is executed by all the nodes including the root node. To ensure this a "virtual root" node ID used to receive output.

All messages have form  $(tag, field_1, \dots, field_n)$ . There are five type messages:

1. Message  $(new, n)$  reports upon detection of a new neighbour  $n$ . During initialisation, all initial neighbours are reported by the protocol.
2. Message  $(fail, n)$  is delivered when failure of node  $n$  has been detected.
3. Message  $(weight, w)$  is a notification about local weight change. The frequency and precision with each it takes place is not specified by the protocol.
4. Message  $(update, n, w, l, p)$  is called an *update vector*. This is the main message, which informs receiver that the BFS tree, rooted in the sender  $n$ , has an aggregate weight  $w$  and that the  $n$ 's level is  $l$  and  $n$ 's parent is  $p$ . The latter is defined if  $n$ 's neighbourhood table has more than one entry.
5. Message  $(timeout)$  is timeout service message and is received upon the timeout.

The protocol uses a *simple rate limitation scheme* to decrease a load on the root node and other nodes. This scheme imposes the upper bound on message rate on each link.

### 2.3.2 The algorithm

The protocol's execution starts with the initialization phase, where all data structures and underlying services are initialized. Thereafter, the protocol goes to an infinite loop, called main loop, given in Fig. 2.3.

Each iteration of the main loop of the algorithm includes three steps:

```

proc gap(Table) = {
/* initialize data structures and services */
  Timeout = 0 ;
  NewNode = null;
  Vector = updatevector(Table);
  NewVector = Vector ;

while true do /* main loop */
  receive
    (new,From) =>
      {newentry(From,Table) ;
       NewNode = From;}
    | (fail,From) =>
      removeentry(From,Table);
    | (update,From,Weight,Level,Parent) =>
      updateentry(Table,From,Weight,Level,Parent);
    | (weight,Weight) =>
      updateentry(Table,self(),Weight,level(self()),parent());
    | (timeout) => Timeout = 1 ;
  end ;
  restoreTableInvariant(Table) ; //see below
  NewVector = updatevector(Table) ;
  if NewNode!= null {
    send(NewNode,newVector) ;
    NewNode = null; }
  if NewVector != Vector && Timeout {
    broadcast(newVector);
    Timeout = 0 ;
    Vector=NewVector ; }
  fi;
od;
};

```

*Figure 2.3: Main loop of the algorithm*

1. Receive a message and change the table accordingly.
2. Execute the procedure *restoreTableInvariant* to update the neighbourhood table consistently with the information received.
3. Multicast the state changes to neighbours if necessary. When a new node has been discovered and added to a neighbourhood table, the update vector must be sent to it to establish the connection. When the update vector have changed and sufficient time has lapsed, the recent update vector is broadcast to the known neighbours.

The system's stabilization is provided by the robust procedure *restoreTableInvariant*, shown in List. 2.4. It is responsible for maintenance of a node's neighbourhood table invariants:

- Each node is associated with at most one entry.
- Exactly one row has status "self", and the node of that entry is node itself (i.e. its id).
- If the table has more than one entry it has a "parent".

```
proc restoreTableInvariant(Table) = {  
  if Table has more than one entry  
  then  
    if Table has no parent  
    then  
      NewParent = nominate node with minimal level(Table) ;  
      row(NewParent,Table).Status = parent ;  
    else if there is node with level less than level(parent(Table))  
    then  
      NewParent = nominate node with minimal level(Table) ;  
      row(parent(Table),Table).Status = peer ;  
      row(NewParent,Table).Status = parent ;  
    fi ;  
    row(self(),Table).Level = row(parent(Table),Table).Level + 1  
  fi ;  
};
```

*Figure 2.4: The procedure restoreTableInvariant()*

- The parent has minimal level among all entries in the neighbourhood table.
- The level of the parent is one less than the level of self.
- Parent is unique if exists.

For the full version of the GAP pseudocode, refer to Fig. [A.1](#).

# Chapter 3

## Case Study with UPPAAL

This chapter describes the application of model checking to protocol verification. In the remainder of this chapter, we review this approach and existing tools, theoretical background on Timed Automata, the background on the UPPAAL tool, and apply these techniques to the GAP protocol. The GAP protocol is a timed asynchronous event-driven system with message-passing communication.

### 3.1 Background

*Model-checking* [14] is an automated technique, by which one can provide a model or specification and validate a required property. A model can be composed of several parts or components. It includes variables with values changing according to specified rules. The values of all variables of a model defines a *system state*. The set of all states reachable by a given system is called *state space*. Model-checking tools perform verification by exhaustively exploring the state space while looking for requirements violation. Verification feasibility depend significantly on the size of state space – large state spaces can quickly consume all available memory and, in the best case, run for a very long time. Thus, the bottleneck is the *state space explosion* problem [4, 14], when the size of the state space grows exponentially to the number of variables and components of the system. Therefore, model-checking does not scale well.

There is a number of tools, widely used for industrial applications, including SPIN [20], SMV [28], UPPAAL [3, 24] and KRONOS [8, 39].

SPIN is a commonly used tool for verifying the correctness of software design in a rigorous and mostly automated fashion. It supports many useful features including a high level language Promela for system specification. However, one weakness of Promela is the absence of support for timers. The time ordering of actions in Promela is implicit and depends on the sequential composition of statements within each one of the component processes, as well as on the unspecified interleaving of statements from different processes. This time relation is non-deterministic and provides no exact time interval that will elapse between two events. This can be a shortcoming when systems are to be verified whose correct functioning depends on timing parameters. Many such examples can be found among communication protocols, such that the duration of timeout intervals is important. For timed systems one could use a real-time extension of SPIN [6, 35].

Because of many successful case studies, the choice of model-checker for this work is UPPAAL, a toolset for validation and model checking of real-time systems.

### 3.1.1 Timed Automata

Since the goal is to verify the GAP protocol, an appropriate formalism suitable for automated verification is needed. Timed automata, proposed by Alur and Dill [2] is such a formalism that extends the well-known notion of finite automata by clocks. In UPPAAL, a system is composed of several timed automata running concurrently.

A *Timed Automaton* [2, 3, 5] is essentially a finite state machine, i.e. a graph consisting of a finite set of nodes or locations and a finite set of labelled edges, extended with real-valued variables. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increase synchronously with the same rate. Clock constraints, i.e. guards on edges, are used to restrict the behaviour of the automaton. A transition represented by an edge can be taken when the clock values satisfy the guard labelled on the edge. Clocks may be reset to zero when a transition is taken.

As an illustration of the concept of time, Fig. 3.1 shows the model of a vending machine example. An automaton, depicted in Fig. 3.1(a), is a timed automaton modelling a vending machine. It has three locations: `wait`, `tea` and `coffee`. If user, an automaton in Fig. 3.1(b), inserts a coin, represented by synchronization `coin!` with `coin?`, then the vending machine offers a tea. However, if user wants a coffee and inserts another coin fast enough, the vending machine offers a coffee. The clock  $y$  of the vending machine is used to detect if the user was fast ( $y < 5$ ) or slow ( $y \geq 5$ ). In both cases – coffee and tea – a user needs two coins, but the choice of the offer depends on the speed of the user.

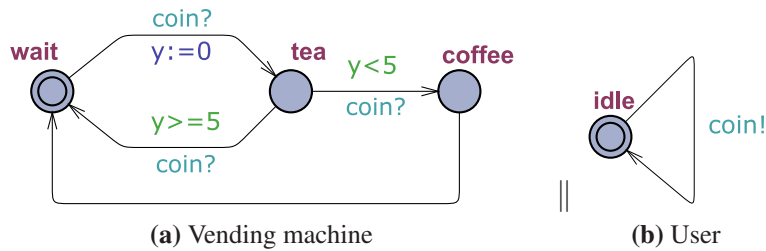


Figure 3.1: The Vending machine example

Let  $C$  be a set of clocks and  $B(C)$  is the set of conjunctions over simple conditions of the form  $x \bullet c$  or  $x - y \bullet c$ , where  $x, y \in C$ ,  $c \in \mathbb{N}$  and  $\bullet \in \{<, \leq, =, \geq, >\}$ . A timed automaton is a finite directed graph annotated with conditions over and resets of non-negative real valued clocks. The following definitions of the syntax and semantics for timed automata are adapted from UPPAAL [3].

**Definition 1 (Timed automaton (TA)).** A *timed automaton* is a tuple  $(L, l_0, C, A, E, I)$ , where  $L$  is a set of locations,  $l_0 \in L$  is the initial location,  $C$  is a set of clocks,  $A$  is a set of actions, co-actions and the internal  $\tau$ -actions,  $E \subseteq L \times A \times B(C) \times 2^C \times L$  is a set of edges between locations with an action, a guard and a set of clocks to be reset, and  $I : L \rightarrow B(C)$  assigns invariants to locations.

The semantics of a timed automaton is defined as a transition system where a state or configuration consists of the current location and the current values of clocks. There are two types of transitions between states. The automaton may either delay for some time (a delay transition), or follow an enabled edge (an action transition).

A *clock assignment* is a function  $u : C \rightarrow \mathbb{R}_{\geq 0}$  from the set of clocks to the non-negative reals. Let  $\mathbb{R}^C$  be the set of all clock assignments. Let  $u_0(x) = 0$  for all  $x \in C$ . Consider guards and invariants as sets of clock assignments, and  $u \in I(l)$  means  $u$  satisfies  $I(l)$ .

**Definition 2 (Semantics of TA).** Let  $(L, l_0, C, A, E, I)$  be a timed automaton. The semantics is defined as a labelled transition system  $(S, s_0, \rightarrow)$ , where  $S \subseteq L \times \mathbb{R}^C$  is a set of states,  $s_0 = (l_0, u_0)$  is the initial state, and  $\rightarrow \in S \times \{\mathbb{R}_{\geq 0} \cup A\}$  is a transition relation such that:

- $(l, u) \xrightarrow{d}(l, u + d)$  if  $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(l)$ , and
- $(l, u) \xrightarrow{a}(l', u')$  if  $\exists e = (l, a, g, r, l') \in E$  s.t.  $u \in g, u' = [r \mapsto 0]u$ , and  $u' \in I(l')$ ,

where for  $d \in \{\mathbb{R}_{\geq 0}, u + d$  maps each clock  $x \in C$  to value  $u(x) + d$ , and  $[r \mapsto 0]u$  denotes the clock assignment which maps each clock in  $r$  to 0 and agrees with  $u$  over  $C \setminus r$ .

Timed automata are often composed into a network of timed automata over a set of clocks and actions:  $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$ ,  $1 \leq i \leq n$ . A location vector is a vector  $\vec{l} = (l_1, \dots, l_n)$ . The invariant functions are composed into a common function over location vectors  $I(\vec{l}) = \bigwedge_i I_i(l_i)$ .  $\vec{l}[l'_i/l_i]$  denotes the vector where the  $i$ th element  $l_i$  of  $\vec{l}$  is replaced by  $l'_i$ .

**Definition 3 (Semantics of a network of Timed Automata).** Let  $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$  be a network of  $n$  timed automata. Let  $\vec{l}^0 = (l_1^0, \dots, l_n^0)$  be the initial location vector. The semantics is defined as a transition system  $(S, s_0, \rightarrow)$ , where  $S \subseteq (L_1 \times \dots \times L_n) \times \mathbb{R}^C$  is a set of states,  $s_0 = (l_0, u_0)$  is the initial state, and  $\rightarrow \in S \times S$  is a transition relation defined by:

- $(\vec{l}, u) \rightarrow (\vec{l}, u + d)$  if  $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(\vec{l})$ , and
- $(\vec{l}, u) \rightarrow (\vec{l}[l'_i/l_i], u')$  if  $\exists l_i \xrightarrow{\tau g r} l'_i$  s.t.  $u \in g, u' = [r \mapsto 0]u$ , and  $u' \in I(\vec{l})$ .
- $(\vec{l}, u) \rightarrow (\vec{l}[l'_i/l_i, l'_j/l_j], u')$  if  $\exists l_i \xrightarrow{c^? g_i r_i} l'_i$  and  $l_j \xrightarrow{c^? g_j r_j} l'_j$  s.t.  $u \in (g_i \wedge g_j), u' = [r_i \cup r_j \mapsto 0]u$ , and  $u' \in I(\vec{l})$ .

As a simplification,  $l_i \xrightarrow{agr} l'_i$  is used instead of  $(l_i, a, g, r, l'_i) \in E$ . This denotes a transition between location  $l_i$  and  $l'_i$ , where  $a$  is a synchronization action,  $g$  is a guard over clocks or integer variables and  $r$  is a set of assignments (or resets) on clocks and variables.

As an example, the vending machine in Fig. 3.1(a) may have the following states:

$(\text{Coffee.wait}, y = 0) \rightarrow (\text{Coffee.wait}, y = 3) \rightarrow (\text{Coffee.tea}, y = 0) \rightarrow (\text{Coffee.tea}, y = 0.5) \rightarrow (\text{Coffee.coffee}, y = 0.5) \rightarrow (\text{Coffee.coffee}, y = 1000)$  and etc.

### 3.1.2 UPPAAL

This section gives a brief presentation of the UPPAAL tool [3] with its internal and query language constructs used in this work.

UPPAAL is a model checker for networks of timed automata, presented in Section 3.1.1. It is used as a tool for modelling, simulation and verification of real-time systems. It is suitable for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real valued clocks. Communication is possible through synchronisation on channels and by exchanging data through shared global variables, as it is shown in Fig. 3.2.

A transition assumed to take no time. The time elapsing in a state depends on its invariant that defines a "deadline" for taking a transition. *Committed* states are left without a time delay before all non-committed states. The process that stays in committed location, marked with "C", doesn't allow other processes to fire their transitions until it reaches non-committed location. Active *urgent* states,

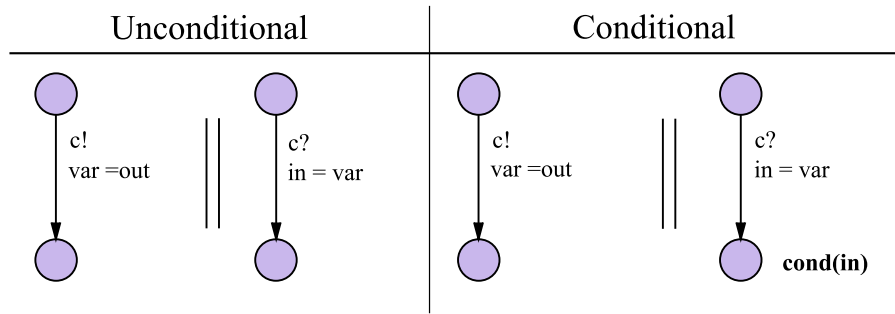


Figure 3.2: Synchronous value passing example in UPPAAL

marked with "U", also prevent time from elapsing but have no priority over non-urgent states. In the actual implementation of UPPAAL, transitions of different nodes do not progress simultaneously, but in an interleaving way.

UPPAAL consists of two main parts: a *graphical-user interface* (GUI) and a *model-checking engine*, also called as a *verifier*. It also includes three important components: a *system editor*, a *description language* and *simulator*.

A system is modelled with the system editor as a number of processes, each of which is a timed automaton, put in parallel. A process is instantiated from a parametrised template. In order to describe a system behaviour, a description language is used. The description language is a non-deterministic guarded command language with real-valued clock variables and data types (Fig. 3.3).

$$\begin{aligned}
 \text{Exp} & ::= \text{ID} \mid \text{NAT} \mid \text{'(' Exp ')'} \mid \text{Exp '[' Exp ']'} \\
 & \mid \text{Exp '++'} \mid \text{'++' Exp} \\
 & \mid \text{Exp '--'} \mid \text{'--' Exp} \\
 & \mid \text{Exp Assign Exp} \mid \text{Unary Exp} \\
 & \mid \text{Exp Binary Exp} \mid \text{Exp '?' Exp ':' Exp} \\
 & \mid \text{Exp '.' ID} \mid \text{Exp '(' Args ')'} \\
 & \mid \text{'forall' '(' ID ':' Type ') Exp} \\
 & \mid \text{'exists' '(' ID ':' Type ') Exp} \\
 & \mid \text{'deadlock'} \mid \text{'true'} \mid \text{'false'} \\
 \text{Args} & ::= [\text{Exp (' , ' Exp)*}] \\
 \text{Assign} & ::= \text{'='} \mid \text{' := '} \mid \text{' += '} \mid \text{' -= '} \mid \text{' *= '} \mid \text{' /= '} \\
 & \mid \text{' %= '} \mid \text{' |= '} \mid \text{' \&= '} \mid \text{' ^= '} \mid \text{' <<= '} \mid \text{' >>= '} \\
 \text{Unary} & ::= \text{'+'} \mid \text{'-'} \mid \text{'!' } \mid \text{'not'} \\
 \text{Binary} & ::= \text{'<'} \mid \text{'<='} \mid \text{'=='} \mid \text{'!='} \mid \text{'>='} \mid \text{'>'} \\
 & \mid \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \text{'%' } \mid \text{'\&'} \\
 & \mid \text{'|'} \mid \text{'^'} \mid \text{'<<'} \mid \text{'>>'} \mid \text{'\&\&'} \mid \text{'||'} \\
 & \mid \text{'<?' } \mid \text{'>?' } \mid \text{'and'} \mid \text{'or'} \mid \text{'imply'}
 \end{aligned}$$

Figure 3.3: Syntax of expressions in BNF

In the current implementation of UPPAAL a system description (or model) consists of a collection of timed automata extended with integer variables in addition to clock variables. The edges of the automata are decorated with three types of labels: a *guard*, expressing a condition on the values of

clocks and integer variables that must be satisfied in order for the edge to be taken; a *synchronization action* which is performed when the edge is taken and finally a number of clock resets and *assignments* to integer variables. All three types of labels are optional. In addition, control nodes may be decorated with so called *invariants*, which are conditions expressing constraints on the clock values in order for control to remain in a particular node.

The simulator of UPPAAL is a validation tool which enables examination of possible dynamic executions of a system during modelling stage. Using the simulator, the system can be simulated according to some random seed. It is very helpful to validate the dynamic behaviour of each design sketch, in particular for fault detection, and later on for debugging the generated diagnostic traces.

The verifier serves to check reachability, safety and liveness properties by exploring the symbolic state space of a system.

## Types

There are four predefined types in the current version of UPPAAL: int, bool, clock and chan. Array and record types can be defined over these and custom types. The BNF is given in Fig. 3.4.

$$\begin{aligned}
 \textit{Type} & ::= \textit{Prefix TypeId} \\
 \textit{Prefix} & ::= \text{'urgent'} \mid \text{'broadcast'} \mid \text{'meta'} \mid \text{'const'} \\
 \textit{TypeId} & ::= \text{ID} \mid \text{'int'} \mid \text{'clock'} \mid \text{'chan'} \mid \text{'bool'} \\
 & \quad \mid \text{'int'} \text{' [' Exp ', ' Exp ' ]'} \mid \text{'scalar'} \text{' [' Exp ' ]'} \\
 & \quad \mid \text{'struct'} \text{' {' Field (Field)* '}} \\
 \textit{Field} & ::= \textit{Type ID Arr * (' , ' ID Arr *) * ' ; ' } \\
 \textit{Arr} & ::= \text{' [' Exp ' ]'} \mid \text{' [' Type ' ]'}
 \end{aligned}$$

**Figure 3.4:** Syntax of types in BNF

## Functions

Functions in UPPAAL can be declared alongside other declarations. The syntax for functions is defined by the grammar shown in Fig. 3.5.

$$\begin{aligned}
 \textit{Fun} & ::= \textit{Type ID (' Params ')} \textit{Block} \\
 \textit{Block} & ::= \text{' {' Decls Stat* '}} \\
 \textit{Stat} & ::= \textit{Block} \mid \text{' ; ' } \mid \textit{Exp} \text{' ; ' } \mid \textit{For} \mid \textit{Iter} \\
 & \quad \mid \textit{While} \mid \textit{DoWhile} \mid \textit{IfStat} \mid \textit{RtStat} \\
 \textit{For} & ::= \text{'for'} \text{' (' Exp ' ; ' Exp ' ; ' Exp ' )'} \textit{Stat} \\
 \textit{Iter} & ::= \text{'for'} \text{' (' ID ':' Type ')} \textit{Stat} \\
 \textit{While} & ::= \text{'while'} \text{' (' Exp ')} \textit{Stat} \\
 \textit{DoWhile} & ::= \text{'do'} \textit{Stat} \text{'while'} \text{' (' Exp ')} \text{' ; ' } \\
 \textit{IfStat} & ::= \text{'if'} \text{' (' Exp ')} \textit{Stat} [ \text{'else'} \textit{Stat} ] \\
 \textit{RtStat} & ::= \text{'return'} [ \textit{Exp} ] \text{' ; ' }
 \end{aligned}$$

**Figure 3.5:** Syntax of functions in BNF

The use of functions considerably simplifies the UPPAAL model. As an example, Fig. 3.6 shows the Node template of GAP as an example of model, implemented without functions.

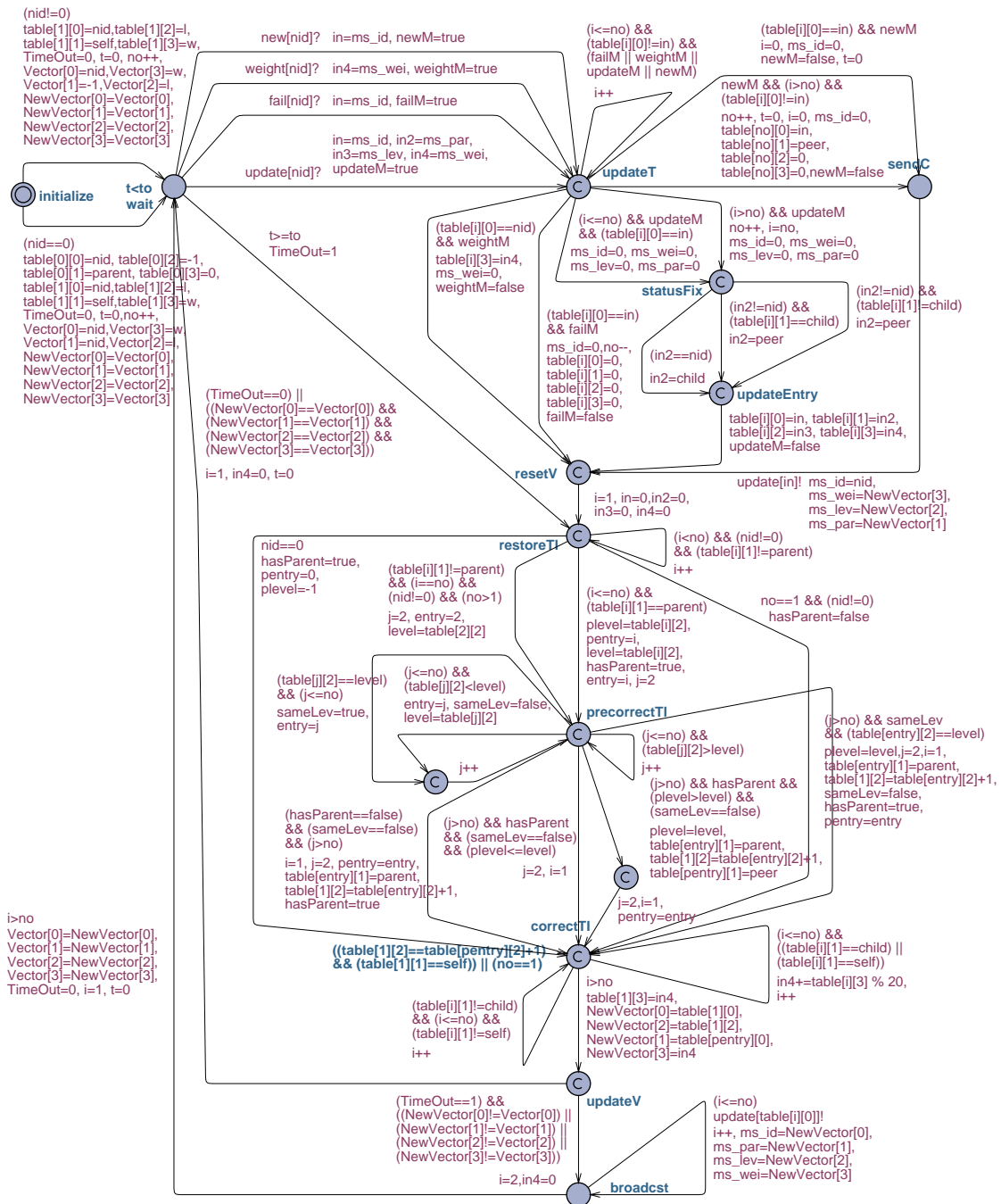


Figure 3.6: GAP's Node template

### Variable reduction

This is the commonly used technique to reduce the size of the state space. Explicit variables reset, when they are not used, helps speed up the verification (Fig. 3.7). Basically, two states that differ only in the values of such variables, are bisimilar. By resetting these variables to the same value, i. e. zero, these two states will become identical, thus, reducing the state space.

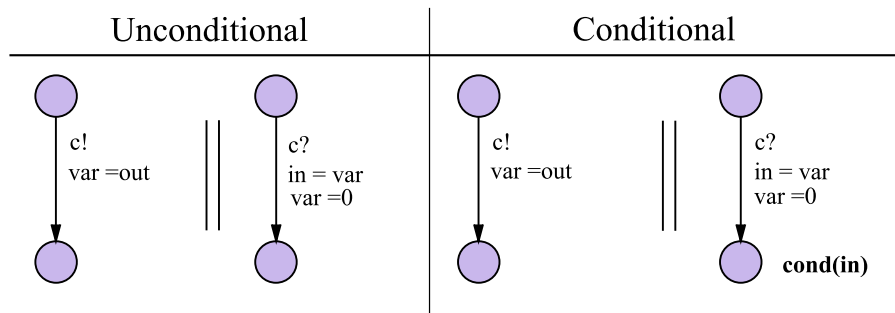


Figure 3.7: Variable reduction

### Query language

The verifier of UPPAAL allows logical queries to verify a system requirements and determine whether a process deadlocks. The tool's response is either "The property is satisfied" or "The property is not satisfied". "The property is maybe satisfied" response is given when the verifier cannot determine the truth value of the property due to the approximations used. For more details, refer to the documentation of UPPAAL.

CTL Formula	UPPAAL form
$A\Box\varphi$ ( $A\Diamond\varphi$ )	$A[\ ]\varphi$ ( $A\langle\rangle\varphi$ )
$E\Box\varphi$ ( $E\Diamond\varphi$ )	$E[\ ]\varphi$ ( $E\langle\rangle\varphi$ )
$\varphi \rightsquigarrow \psi$	$\varphi \dashrightarrow \psi$
$\neg\varphi$	not $\varphi$
$\varphi \wedge \psi$	$\varphi$ and $\psi$
$\varphi \vee \psi$	$\varphi$ or $\psi$
$\varphi \Rightarrow \psi$	$\varphi$ imply $\psi$

Table 3.1: Correspondence between CTL and UPPAAL query language syntax

The requirement specification language of UPPAAL is a simplified version of Computation Tree Logic (CTL), used to express the property of an automata. Table 3.1 shows the relationship between CTL syntax and UPPAAL's query language syntax. The language formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers describe the branching structure in the searching and include two quantifiers: A ("for all computation paths") and E ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have a specified property, respectively.

The temporal operators describe properties of a path through the searching. The main operators are as following:

1. The *globally* operator:  $G\varphi$  is true at time  $t$  if  $\varphi$  holds at all  $t' \geq t$ . This operator is denoted by  $[\ ]$  in UPPAAL.
2. The *future* operator:  $F\varphi$  is true at time  $t$  if  $\varphi$  holds at some  $t' \geq t$ . This operator is represented by  $\langle\rangle$  in UPPAAL.

Hence, the formulas should be one of the following forms:

- Invariantly  $\varphi$ :  $A[\ ]\varphi$ . This is the same as not  $E\langle\rangle$  not  $\varphi$ . An example is shown in Fig. 3.9(a).

- Possibly  $\varphi$ :  $E\langle\rangle\varphi$ . An example is shown in Fig. 3.8.
- Always eventually  $\varphi$ :  $A\langle\rangle\varphi$ . This is equivalent to  $\text{not } E[]\text{ not } \varphi$ . An example is shown in Fig. 3.10(a).
- Potentially always  $\varphi$ :  $E[]\varphi$ . An example is shown in Fig. 3.9(b).
- $\varphi$  leads to  $\psi$ :  $\varphi \dashrightarrow \psi$ . This is the shorthand for  $A[](\varphi \text{ imply } A\langle\rangle\psi)$ . An example is shown in Fig. 3.10(b).

Here  $\varphi, \psi$  are local properties that can be checked locally on a state, i.e. boolean expressions over predicates on locations and integer variables, and clock constraints.

In UPPAAL, a deadlock of the system is expressed using a special keyword `deadlock`. Due to limitations in current version of UPPAAL, this keyword can be used only in either  $A[]$  or  $E\langle\rangle$  form. The state property `deadlock` evaluates to *true* for a state  $(l, u)$  if and only if for all  $d \geq 0$  there is no action successor of  $(l, u + d)$ .

The classification of path formulas includes *reachability*, *safety* and *liveness* properties.

**Reachability Properties.** A property  $\varphi$  holds if there exists a path starting from initial state, such that  $\varphi$  is eventually satisfied along that path. These properties are often used to ensure of basic behaviour of the system to be verified and expressed using the path formula  $E\Diamond\varphi$ , shown in Fig. 3.8.

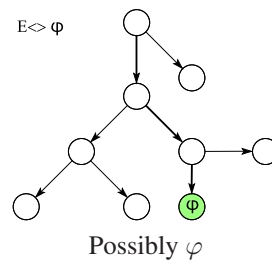


Figure 3.8: Reachability property example

**Safety Properties.** A safety property ensures that some property  $\varphi$  will never hold. In UPPAAL, a safety property is formulated as a property  $\varphi$  that holds for every reachable state. Hence, property  $\varphi$  should be true in all reachable states with the formula  $A[]\varphi$ , whereas  $E[]\varphi$  claims that there exists a maximal path that is either infinite or with the last state with no outgoing transitions, such that  $\varphi$  is always true. Both examples are shown in Fig. 3.9.

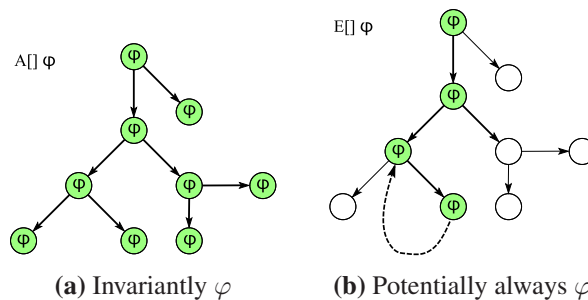


Figure 3.9: Safety property examples

**Liveness Properties.** This property checks whether some property  $\varphi$  will eventually hold. The simplest form of this property expresses with the formula  $A\Diamond\varphi$ . Another form is the *leads-to* or *response* property, expressed by  $\varphi \rightsquigarrow \psi$  and holds if and only if whenever  $\varphi$  holds, eventually  $\psi$  will hold as well.

In addition, UPPAAL allows to check *bounded liveness properties*, that guarantees not only to hold eventually but also within some specified upper time limit: formula  $\varphi \rightsquigarrow_{\leq t} \psi$  expresses that whenever the state property  $\varphi$  holds then the state property  $\psi$  holds within at most  $t$  units of time thereafter. There exists two variations of the bounded liveness property – reduction for *unbounded leads-to* and safety property. In the first case, the bounded liveness property is obtained by verifying  $\varphi \rightsquigarrow (\psi \wedge x \leq t)$ , where  $x$  is an additional clock which is reset whenever  $\varphi$  holds. In the second version, bounded liveness property is obtained by verifying the safety property  $A\Box(b \Rightarrow \psi \wedge x \leq t)$ , where whenever  $\varphi$  starts to hold a boolean variable  $b$ , initialized to false, is set to true and an additional clock  $x$  is reset.

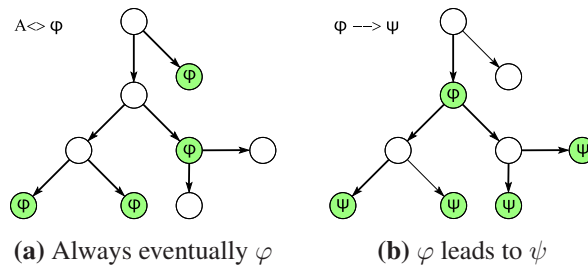


Figure 3.10: Liveness property examples

### Symmetry reduction

Symmetry reduction is a technique to reduce the resource requirements for model checking algorithms to handle the state space explosion problem, and it has been implemented in model checkers such as Mur $\varphi$  [10, 21], SMV [28], Spin [7, 20] and UPPAAL.

In particular, a new type constructor *scalarset* extends the description language of UPPAAL allowing symmetric data types to be syntactically defined. The elements of scalarsets use name equivalence.  $n$  process identifiers can be modelled using a scalarset of size  $n$ : `scalar[n]`. One can declare arrays over the elements of scalarsets as follows:

```
typedef scalar[5] proc_id;
int node[proc_id];
process Node (const proc_id pid)
```

Comparison between scalars and integers is not allowed. Initialisers for scalars, however, are supported, e.g. `proc_id node = 1;`. The instantiation of a template and declaration of all process in the system (size of scalarset) is as follows:

```
Procs = forall i in proc_id : Node(i);
system Procs;
```

The `forall` construct iterates over all elements of a declared scalarset type. In this case the iteration is over `proc_id` and a set of instances of the template `P` is constructed and bound to `Procs`.

More information about this constructor can be found in [1]. The soundness of symmetry reduction for UPPAAL has been proved by Hendriks [18]. Hendriks *et al.* [19] describe a prototype extension of UPPAAL with symmetry reduction.

### Template set mechanism

*Template set* is a mechanism for process template instantiation as an array of processes. It is used to declare a template over a set. Unlike symmetry reduction, the mechanism is not limited to scalarsets.

The mechanism draws some similarities with the *RuleSet* construction of the *Mur $\phi$*  tool [1]. It can be declared over integer types starting with a range starting at 0. The following is an example of syntax for template sets:

```
process Node(const id_t nid, id_t &i, const int &w, const int &l) { ... }
```

The first parameter is an implicitly given parameter that gives rise to an array of processes. This argument can be specified when instantiating the template as follows:

```
//system global code
const int n = 6;
typedef int[0,n-1] id_t;
```

The resulting template is later instantiated into a process by listing it in the system line.

```
//system declaration
id_t ar;
node (const id_t id) = Node (id, ar, 1, 0);
system node;
```

## 3.2 Modelling the GAP Protocol

This section describes details of the UPPAAL model of the GAP protocol. The UPPAAL model has been constructed to reflect the original protocol description as closely as possible. The advantage of this idea is that the relationship between the protocol description and its model is easily understood, but the drawback is that the model is too complex, and, thus, the system faces the state space explosion problem described in Section 3.1. Note that the GAP model presented here is based on the specifications and features of UPPAAL 4.0.

### 3.2.1 Architecture of the model

Fig. 3.11 shows a model of a node executing the GAP protocol. A network consists of an arbitrary number of processes or nodes, linked together, each of which is represented by `Node` template. For each node, there is an automaton `Buffer` that realises a buffer. It emulates asynchronous message-passing, the communication model of GAP, by storing one message received from another `Node` process.

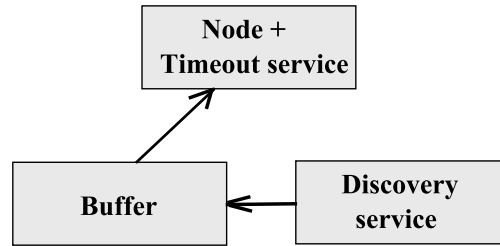


Figure 3.11: Architecture of a node in the GAP model in UPPAAL

In addition, a node is equipped with Discovery and Timeout services. The Discovery service is responsible for new neighbours discovery. Timeout service, responsible for sending of "timeout" messages, is "injected" into the Node automaton.

This thesis work considers only initial stabilization of the system, i.e. initial construction of a BFS tree from the underlying network graph. Thus, three services, the presence of which is assumed by the protocol, are not included in the simplified model of GAP: *weight subscription service*, *failure detection service* and *message delivery service*. This is done to minimize system's state space.

The weight subscription service sends "weight" message to its node about change of the local weight. The failure detection service is required to inform its node about a node failure by sending "fail" messages. The message delivery service is not included to the model, since we assume that message delivery is reliable, i.e. no message loss.

List. B.4 gives a system definition and process assignments used in the present case study.

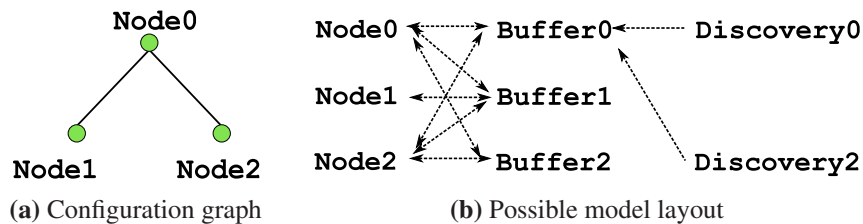


Figure 3.12: Model for three nodes

Fig. 3.12 shows an example of the system of three nodes. Fig. 3.12(b) depicts the interdependences of all automata for the system, configuration of which is shown in Fig. 3.12(a). The automate connected by the direct line communicate with each other through one or more synchronisation channels. The arrows on the figure show the direction of the communication between automata.

### 3.2.2 Timing Information

In UPPAAL, time elapses for all processes synchronously as long as no invariant is violated. All clock values increase by the amount of time elapsed. Transitions normally assumed to take no time, but this happens in the GAP system. Thus, all timing information in the system can be described as follows:

- *Rate limitation scheme*: "timeout" messages are periodically sent by the Timeout service of each node. This is so called *periodical heartbeats* of the system. The period is defined by the global variable  $t_o$ .



	Index			
Variable	0	1	2	3
Vector [ <i>i</i> ], NewVector [ <i>i</i> ]	ID	parent	level	weight
table[#entry] [ <i>i</i> ]	status	level	weight	-

Table 3.2: Neighbourhood table's and vector arrays' "Index-Field" correspondence

ID 0 and level  $-1$  always holds "entry" 0 in the neighbourhood table. If this node doesn't exist in a neighbourhood table, the "status" field will be 0.

The system starts with initial values, assigned by the function `startInit()`. It checks an ID of the started node. If a node is an arbitrary node, other than the root node, this function will create one entry of its local neighbourhood table with information about itself and initialize all flags according to the protocol's specification. If a node is the root node, in addition, this function includes the "virtual root" information to the entry  $i = 0$  and changes the boolean variable `hasP` to `true` to record that the node has a parent. Also, it changes flag `one` to `false` to indicate that there is more than one entry in its neighbourhood table.

Each node has an access to the global variable `cnt` and subtracts 1, firing the transition `initialize`  $\rightarrow$  `wait`. This is how the function `count()` works. It is used to capture the moment when a node enters the main loop and starts listening for incoming messages at the location `wait`.

From time to time a node exchanges the information with neighbours from its neighbourhood table by sending them appropriated messages. As it was mentioned in Sect. 3.1.2, processes communicate by synchronisation via channels and pass data via the shared global variables. In this model, a Node process communicates via `new[nid]`, `fail[nid]`, `update[nid]` and `bweight[nid]` channels, depending on the type of "message" and reads the global variables `ms_id`, `ms_wei`, `ms_par`, `ms_lev` (see Table 3.3).

field	ID	parent	level	weight
shared variable	<code>ms_id</code>	<code>ms_par</code>	<code>ms_lev</code>	<code>ms_wei</code>

Table 3.3: Global shared variables

When a "new" message arrives, by executing the function `newadd()`, a node checks its neighbourhood table to find out whether the node with ID, equal to the one in the message, exists in its table. If this entry exists, the receiver resets shared variables with `resetSharVar()`, executes the procedure `resTabInv()` and sends an update vector to the node. If not, the function `newadd()` adds a new node with ID from the message and assigns default fields to the new entry.

`resetSharVar()` is used to reduce the size of the state space by resetting variables, that are no longer needed. This technique is called *variable reduction* and helps to speed up the verification [3].

The function `failrem()` processes "fail" messages. It implements the function `removeentry(n)` of the protocol and resets the corresponding entry in the neighbourhood table. That is, it checks if the "status" column is not equal to 0 and resets all columns. When "update" message arrives, Node executes the function `updateadd()` to translate parent ID of the sender into its status with respect to the current node. The processed information is added to the neighbourhood table.

A Timeout service is merged with the Node process. A local clock  $t$  keeps track of the time between two consecutive "heartbeats of the system", as it is in the "rate limitation scheme" (Sect. 3.2.2). A Node automaton must spend exactly  $t$  time units in the location `wait` before it fires the transition

to location `resTI`, assigning `Timeout = 1`. This transition is also guarded by the boolean function `go()`.

The main procedure, `restoreTableInvariant`, is implemented by the function `resTabInv()`. If there is more than one entry in the neighbourhood table, the function will execute an algorithm according to List. A.1. First, it checks whether the node knows its parent, i.e. `flag hasP`. If `hasP == false`, the function will search for the first non-empty entry, assign it as a parent and sets `hasP` to `true`. If `hasP == true`, the function will search for minimal level by comparison of other entries' level with the parent's one, excluding itself and empty entries. Boolean function `invTable()` in location `correctTI` ensures that an invariant `table[nid+1][1] = plevel+1` holds after execution of the function `resTabInv()`.

Note that an evidence of the empty entry of the neighbourhood table is 0 in the "status" column – `table[i][0] = 0`.

The function `genUpVector()` aggregates the weight of the node and its children using the variable `aggr` and updates `NewVector[i]` array:

```
NewVector[0] = nid;
NewVector[1] = p;
NewVector[2] = table[nid+1][1];
NewVector[3] = aggr;
```

In order to use a small range for the integer variables `aggr` and `NewVector[3]`, the parity function is used as an aggregation function. The parity of an integer is its attribute of being even or odd. Thus, the function is computed as follows: `aggr = (aggr + 1) % 2`.

Before sending an "update" message, Node should prepare the update vector, by executing the function `prepUpVec()`. Basically, it is an assignment of `NewVector[i]` array to the global shared variables (see Table 3.2 and Table 3.3). The flag `newVectorEq()` checks whether `NewVector[i]` differs from `Vector[i]`, for  $i = \overline{0,3}$ . Finally, the function `arraysMerge()` does the assignment `Vector[i] = NewVector[i]`.

To overcome the problem with synchrony, e.g. two nodes try to send an update vector to each other when their buffers are full, flags `esc[nid]` and `bc` are used. That is, when a node has a full buffer, it changes the flag `esc[nid]` to `true`, where `nid` is its ID. This flag enables an escape transition `broadcast -> wait` to prevent a deadlock. In case, a broadcast is cancelled, a node sets the flag `bc` to `true` in order to resend an update vector later.

### 3.2.4 Discovery template

An automaton `Discovery`, Fig. 3.14, is a template for a local discovery service of a node, with the parameters `sid` and `did`:

```
process Discovery(const id_t &did, const id_t &sid) { ... }
```

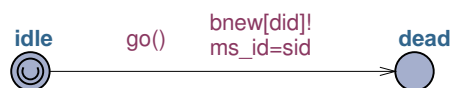


Figure 3.14: Discovery template

Each instance of the `Discovery` template models a link between two neighbouring nodes. The parameters of the template are hardcoded, e.g.:

```

Discovery0:=Discovery(0,1);
Discovery1:=Discovery(1,3);
Discovery2:=Discovery(0,2);
Discovery3:=Discovery(1,4);
Discovery4:=Discovery(1,2);
Discovery5:=Discovery(4,5);
Discovery6:=Discovery(3,4);

```

The service sends "new" messages through `bnew[did]` channel to inform a node with ID `did` about a new neighbour with ID `sid` passed via shared variable `ms_id`. The flag `go()` is used to initialize the service only after all Node processes start their execution.

In general, the *failure detection service* can be modelled similar to the discovery service. The former sends a `fail` message to its node with ID of failed node, one of the node's neighbour. Thus, it should synchronize on the channel `bfail[did]`.

### 3.2.5 Buffer template

The GAP protocol assumes asynchronous communication between the nodes in a network. UPPAAL supports only synchronous communication. For each node there is a Buffer process, illustrated by Fig. 3.15, which emulates asynchronous communication of the system in UPPAAL.

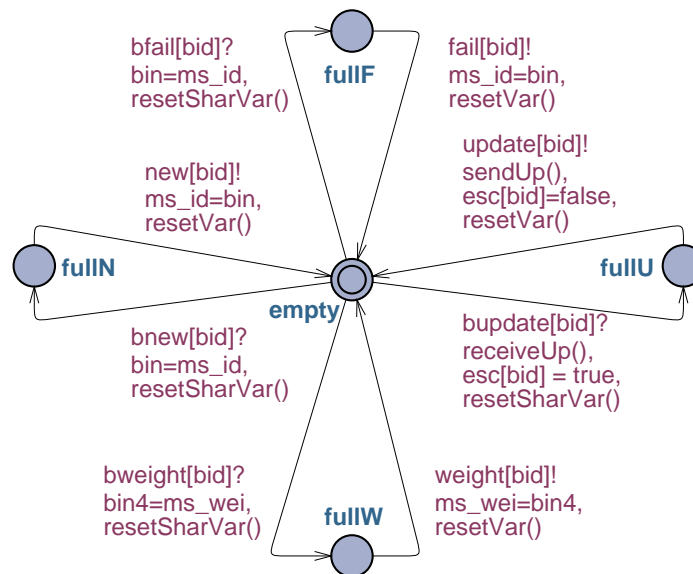


Figure 3.15: Buffer template

The automaton consists of five locations and eight transitions. This template is parametrised with one constant, ID `bi`:

```

process Buffer(const id_t bid, id_t &bi) { ... }

```

It is 1-place buffer that stores the message when it arrives to the node from another node. The main locations are: `empty`, `fullN`, `fullF`, `fullW` and `fullU`. An automaton Buffer can store only one "message" type: "new", "fail", "weight", "update". In fact, Buffer receives a "message" by synchronizing with a Node process on `bnew[bid]`, `bfail[bid]`, `bupdate[bid]` or `bweight[bid]`

channels and by reading the global variables `ms_id`, `ms_wei`, `ms_par`, `ms_lev`. Each shared variable implements one message field, shown in Table 3.3. Buffer receives and stores a message until its Node process retrieves this message by synchronizing via `new[bid]`, `fail[bid]`, `update[bid]` or `weight[bid]` channels. If a buffer is full, it sets flag `esc[bid]` to true to show that buffer is occupied and escape transition is possible.

Similar to Node, Buffer has functions `resetSharVar()` and `resetVar()` to reduce the size of the state space by resetting variables, that are no longer needed.

### 3.3 Verification

This section presents the results of the verification of the GAP model. The UPPAAL model checking system can verify safety, liveness, bounded liveness and reachability properties of a system.

The finite model achieved after the modelling stage is still complex with respect to the memory cost. Thus, a restriction to a small number of processes is required in order to check correctness with UPPAAL. We restrict ourselves up to four Node processes. Moreover, some properties are difficult to check even for four nodes as it requires a lot of memory space as well as time. Hence, a full evidence of the correctness of the system cannot be provided.

The verification of the GAP model has been run on two computers:

- Intel Xeon CPU 2.40 GHz, 3GB RAM with OS Windows Server 2003 R2
- Intel Pentium 4 CPU 2.66 GHz, 512MB RAM with OS Windows XP

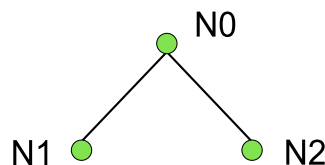
To remind, the GAP model presented here is based on the specifications and features of UPPAAL 4.0.0.

#### 3.3.1 Configurations

Every verified configuration is a tree graph with Node processes as its vertices. All links, each represented by one Discovery service, between the connected neighbours form a tree, i.e. these links are the edges of the graph. We have considered the following configurations with different number of nodes, running in the system:

1. A system with three nodes (Fig. 3.16). The edges of the tree are the following:

```
Discovery0:=Discovery(0,1);
Discovery2:=Discovery(0,2);
```



*Figure 3.16: Tree graph with three nodes (Config. 1)*

2. A system with three nodes (Fig. 3.17). The edges of the tree are the following:

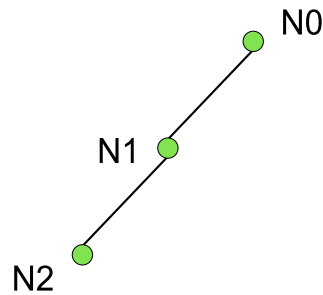


Figure 3.17: Tree graph with three nodes (Config. 2)

```

Discovery0:=Discovery(0,1);
Discovery3:=Discovery(1,2);

```

3. A system with three nodes (Fig. 3.18). This case is similar to the previous one, except one link:

```

Discovery0:=Discovery(0,1);
Discovery2:=Discovery(0,2);
Discovery3:=Discovery(1,2);

```

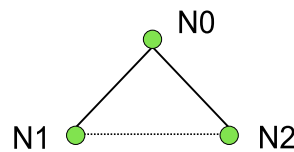


Figure 3.18: Tree graph with three nodes (Config. 3)

4. A system with four nodes and three links, given in Fig. 3.19. This tree consists of the following edges:

```

Discovery0:=Discovery(0,1);
Discovery1:=Discovery(1,3);
Discovery2:=Discovery(0,2);

```

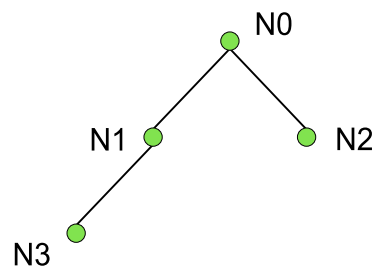


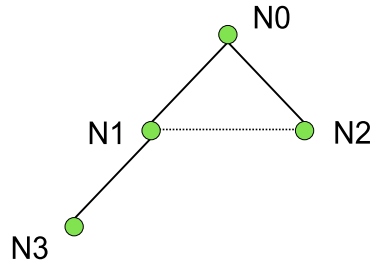
Figure 3.19: Tree graph with four nodes (Config. 4)

5. A system with four nodes as it is shown in Fig. 3.20. This tree has the following edges:

```

Discovery0:=Discovery(0,1);
Discovery1:=Discovery(1,3);
Discovery2:=Discovery(0,2);
Discovery3:=Discovery(1,2);

```



*Figure 3.20: Tree graph with four nodes (Config. 5)*

The dashed lines on all three figures represent the links between the nodes in the underlying graph, but these links are not a part of BFS tree.

We have verified only the initial phase of the protocol’s stabilization, where a BFS tree is constructed on top of the underlying network graph. This doesn’t include a case of nodes failure and, thus, a model of the failure detection service has not been added into our GAP model. Moreover, we have assigned all nodes unit weight and exclude weight subscription service from the model.

### 3.3.2 Validation

As it was mentioned before, the protocol’s model has been constructed to reflect the original protocol description as closely as possible. Yet, with all reduction techniques and mechanisms implemented in order to reduce the complexity and to overcome problems with synchrony, the GAP model differs from the protocol’s description and requires a validation. As the protocol has two main tasks to accomplish, there are two ways of validating the model correctness.

The first approach is to check whether the aggregation function outputs on each node are correct. Another way to validate our model is to check if a BFS tree is constructed. Both approaches could be done for each configuration by the test functions `bfsT()` and `isAggr()` as it is shown in the following sections.

#### Configuration 1

```

void bfsT(){
    output = false;
    if (nid == 0) {
        if ((table[0][0]==parent) && (table[2][0]==child) &&
            (table[3][0]==child)) {output = true;}
    }
    if (nid == 1 || nid == 2) {
        if (table[1][0]==parent) {output = true;}
    }
}

```

```

void isAggr(){
    output = false;
    if (nid == 0 || nid == 1 || nid == 2) {
    if (NewVector[3] == 1) {output = true;}
    }
}

```

### Configuration 2

```

void bfsT(){
    output = false;
    if (nid == 0) {
    if ((table[0][0]==parent) && (table[2][0]==child))
    {output = true;}
    }
    if (nid == 1) {
    if ((table[1][0]==parent) && ((table[3][0]==child)
    || (table[3][0]==peer))) {output = true;}
    }
    if (nid == 2) {
    if (table[2][0]==parent) {output = true;}
    }
}

```

```

void isAggr(){
    output = false;
    if (nid == 0 || nid == 1) {
    if (NewVector[3] == 0) {output = true;}
    }
    if (nid == 2) {
    if (NewVector[3] == 1) {output = true;}
    }
}

```

### Configuration 3

```

void bfsT(){
    output = false;
    if (nid == 0) {
    if ((table[0][0]==parent) && (table[2][0]==child)
    && (table[3][0]==child)) {output = true;}
    }
    if (nid == 1) {
    if ((table[1][0]==parent) && (table[3][0]==peer))
    {output = true;}
    }
    if (nid == 2) {

```

```

if ((table[1][0]==parent) && (table[2][0]==peer))
  {output = true;}
}
}

```

```

void isAggr(){
  output = false;
  if (nid == 0 || nid == 1 || nid == 2) {
  if (NewVector[3] == 1) {output = true;}
  }
}

```

#### Configuration 4

```

void bfsT(){
  output = false;
  if (nid == 0) {
  if ((table[0][0]==parent) && (table[2][0]==child)
    && (table[3][0]==child)) {output = true;}
  }
  if (nid == 1) {
  if ((table[1][0]==parent) && (table[4][0]==child))
    {output = true;}
  }
  if (nid == 2) {
  if (table[1][0]==parent) {output = true;}
  }
  if (nid == 3) {
  if (table[2][0]==parent) {output = true;}
  }
}

```

```

void isAggr(){
  output = false;
  if (nid == 0 || nid == 2 || nid == 3) {
  if (NewVector[3] == 1) {output = true;}
  }
  if (nid == 1) {
  if (NewVector[3] == 0) {output = true;}
  }
}

```

#### Configuration 5

```

void bfsT(){
  output = false;

```

```

if (nid == 0) {
if ((table[0][0]==parent) && (table[2][0]==child)
&& (table[3][0]==child)) {output = true;}
}
if (nid == 1) {
if ((table[1][0]==parent) && (table[3][0]==peer)
&& (table[4][0]==child)) {output = true;}
}
if (nid == 2) {
if ((table[1][0]==parent) && (table[2][0]==peer))
{output = true;}
}
if (nid == 3) {
if (table[2][0]==parent) {output = true;}
}
}

void isAggr(){
output = false;
if (nid == 0 || nid == 2 || nid == 3) {
if (NewVector[3] == 1) {output = true;}
}
if (nid == 1) {
if (NewVector[3] == 0) {output = true;}
}
}
}

```

For validation, the liveness property can be used as follows:

```

(forall (i : id_t) node(i).wait and node(i).TimeOut==1
and buffer(i).empty and node(i).bc==0 and
forall (j : int[0,3]) node(i).Vector[j] == node(i).NewVector[j])
--> (forall (i : id_t) node(i).output==1)

```

Table 3.4 outlines the technical details obtained from the validation.

# nodes	Config.	Time (min)	Memory (MB)	UPPAAL Options			
				SO	SSR	SSRep	DT
3	1	1/6	24	BFS	DBM	Aggressive	Some
3	2	1	111	BFS	DBM	Aggressive	Some
3	3	16	1237	BFS	DBM	Aggressive	None
4	4	270	1970	DFS	CDS	Aggressive	None
4	5	1310	2656	DFS	CDS	Aggressive	None

*Table 3.4: BFS construction bfsT()*

### 3.3.3 Requirements

All requirements have been verified to be true for all configurations, described in the previous section, though a verification process is both time- and memory- consuming.

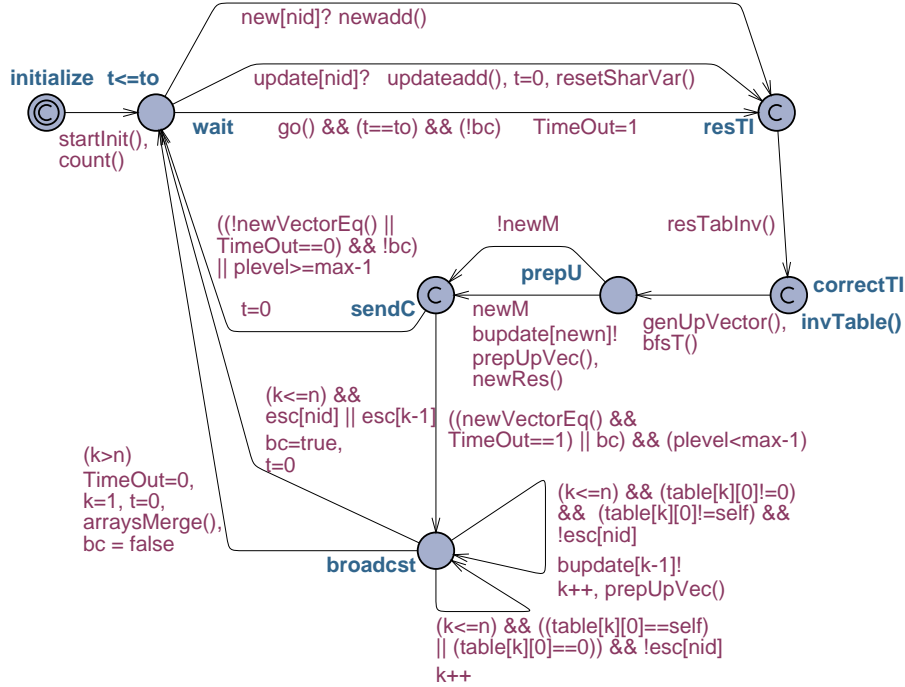


Figure 3.21: Simplified Node template

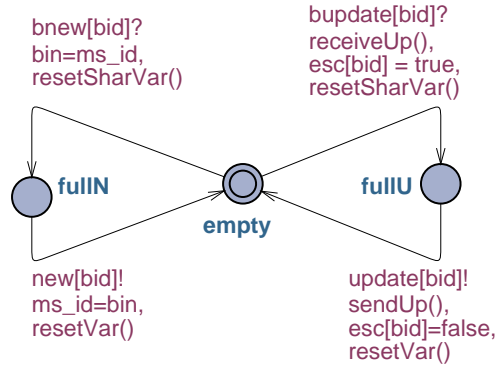


Figure 3.22: Simplified Buffer template

To simplify the GAP model, the failure detection service and weight subscription service are excluded from the model verified (Fig. 3.21 and Fig. 3.22).

UPPAAL’s ”Option” menu contains settings to control the behaviour of the model-checker.

A list of abbreviations used in this section is the following: Search Order – SO, State Space Reduction – SSR, State Space Representation – SSRRep, Diagnostic Trace – DT, Compact Data Structures – CDS, Difference Bound Matrices – DBM.

Properties with a fairness cannot be verified in UPPAAL. But in our model there might be a situation when two processes are executed forever without allowing the third process to be executed. Suppose the third process is the root node. Thus, according to the protocol, two nodes will send an

update vector to each other and increase the level. This case causes a serious problem, since both nodes can increase their level forever. To overcome this problem, we put restrictions on transitions `sendC --> broadcast` and `sendC --> wait`.

### Deadlock freeness

The GAP system should be deadlock free. That is, the system never ends up in a state where it cannot perform any action. As the UPPAAL model is composed of concurrent processes each component of the system should be deadlock free on all paths.

This can be formulated in UPPAAL using the primitive deadlock (Sect. 3.1.2) as expression:

```
A[] not deadlock
```

For further technical information, see Table 3.5.

# nodes	Config.	Time (min)	Memory (MB)	UPPAAL Options			
				SO	SSR	SSRep	DT
3	1	1/2	26	BFS	DBM	Aggressive	None
3	2	1	73	BFS	DBM	Aggressive	Some
3	3	28	1234	BFS	DBM	Aggressive	None
4	4	210	1677	BFS	DBM	Aggressive	None
4	5	1440	2104	DFS	CDS	Aggressive	None

*Table 3.5: Deadlock verification*

### Liveness

The liveness condition, which we verified, expresses a convergence property of the protocol. The liveness condition is the following: it is always possible for the system to get to a state `wait` in which there will be no activity (for all  $j$  `Vector[j] == NewVector[j]`) except `node(i).Timeout` for any `id i` of a node. This requirement expresses the stabilization condition of the BFS tree in terms of the current system.

Bounded liveness is a property that holds within a certain time bound. In general, we are interested in a property: the fact that the protocol converges in the bound of the convergence time. The property is reformulated as a reachability property – the system converges providing that all nodes are in the listening state (i.e. `node(i).wait`), all buffers are empty (i.e. `buffer(i).empty`), no broadcast activity cancelled (i.e. `bc == 0`) and there is no change in the update vector, i.e. `Timeout==1` and for all  $j$  `Vector[j] == NewVector[j]`. This system's state should lead to the state where `output == 1`, which means that BFS tree has been constructed correctly and the time is `GTime>=(n-1)*to`.

```
(forall (i : id_t) node(i).wait and node(i).Timeout==1 and
  buffer(i).empty and node(i).bc==0 and forall (j : int[0,3])
  node(i).Vector[j] == node(i).NewVector[j]) -->
(forall (i : id_t) node(i).output==true) and GTime>=(n-1)*to
```

The technical information on the verification is shown see Table 3.6.

In order to find fastest solution UPPAAL's "Diagnostic Trace:Fastest" option can be used. However, this solution for showing the optimal convergence time is not scalable as verification time grows rapidly.

# nodes	Config.	Time (min)	Memory (MB)	UPPAAL Options			
				SO	SSR	SSRep	DT
3	1	1/12	7	BFS	CDS	Aggressive	Some
3	2	1/2	66	BFS	DBM	Aggressive	None
3	3	12	584	BFS	CDS	Aggressive	None
4	4	205	1306	DFS	CDS	Aggressive	None
4	5	1490	2144	DFS	CDS	Aggressive	None

*Table 3.6: Liveness verification*

## Safety

Safety properties are some properties that are required to always hold, i.e., they are invariants. In GAP, a safety condition holds for all  $i$  of  $\text{node}(i)$ . This condition is an invariant property given in the GAP protocol. The property states that it is always the case that whenever a node is initialized, its level is the level of its parent increased by 1 or this node has no parent. This invariant property can be expressed as:

```
A[] forall (i : id_t)
    node(i).wait imply ((node(i).NewVector[2]==node(i).plevel+1)
                        || (node(i).one))
```

Here, the boolean variable `one` indicates whether a node has only one entry in its local neighbourhood table.

An addition information related to the verification is given in Table 3.7.

# nodes	Config.	Time (min)	Memory (MB)	UPPAAL Options			
				SO	SSR	SSRep	DT
3	1	1/12	8	BFS	CDS	Aggressive	Some
3	2	1/2	65	BFS	DBM	Aggressive	None
3	3	26	1238	BFS	DBM	Aggressive	None
4	4	270	1733	DFS	CDS	Aggressive	None
4	5	1452	2593	DFS	CDS	Aggressive	None

*Table 3.7: Safety verification*

## 3.4 Discussion

This chapter has examined the UPPAAL model for the GAP protocol. First, we have introduced a concept of timed automata and the basics of the UPPAAL tool to give an insight of the model. Thereafter, the model, presented in Sect. 3.2, has been designed, validated and verified for the defined properties to check if it meets all requirements of the actual protocol description.

Several difficulties have been experienced in the designing and verification of the UPPAAL model. First of all, in UPPAAL each process is a (non-deterministic) timed automaton and inherits the non-deterministic behaviour, whereas a node behaviour in GAP is deterministic. For instance, the rate limitation mechanism is tricky to implement as a periodical event.

Another problem is a communication mechanism in UPPAAL. Value passing via global shared variable can be a limitation when a number of message fields grows. In GAP, a number of message fields is small enough to make the protocol suitable to model in UPPAAL. However, there is no "good" way to pass a list or an array, e.g. `ThresholdList` in the TCA-GAP protocol (see List. A.2).

Also, UPPAAL has no primitive for multicast. In our model, a multicast is implemented as a traversal of the list of a node's neighbours and synchronization on channels. As an option, UPPAAL's primitive broadcast is possible to be used with some restriction on the receiver's ID.

Several problems related to query language have been faced during the verification process. For instance, the fairness constraints can be expressed in the linear temporal logic (LTL), but not in CTL. It is also complicated to express the property that will hold at some point in future and henceforth.

The current study has shown that the straightforward model of GAP is too complex – the final model is hard to understand and verify. UPPAAL as well as other model-checking tools has a problem of a state space explosion – a state space grows quickly with an increasing number of interleaving components. The verification process becomes time and memory consuming or impossible. Therefore, we have tried several techniques to reduce the complexity of the model. The details on the implementation have been discussed in the previous sections. In addition, a number of nodes in the system and, thus, the possible scenarios have been limited to reduce the state space.

The same problem arises for the TCA-GAP protocol, where each neighbourhood table has an additional (compared to GAP) column – local thresholds (see List. A.2). Thus, the amount of data stored by each neighbourhood table increases that may lead to less restricted configurations possible to verify.

Few techniques, implemented in the current version of the tool, are inadmissible for the current case study. For example, performance analysis of Alternative Bit Protocol using symmetry reduction [18, 19] has shown the efficiency of the feature. However, this mechanism is not applicable for the GAP protocol due to the limitations of `scalarset` – the function which maps a node identifier to an index of the entry in a neighbourhood table is not possible as scalars use name equivalence. Another example, an abstraction techniques (patterns) for UPPAAL [3] can reduce the GAP protocol to the model of another protocol. That is, by omitting the neighbourhood table of all nodes the protocol is reduced to a model of a BFS tree construction algorithm, similar to the one considered by Dolev *et al.* [11] and Gartner [15]. On another hand, our verified model can be reduced to a simplified one by omitting a weight information and aggregation computation.

In spite of these difficulties, the graphical interface of the tool helps to ease the process of modelling at the earlier stage of modelling as well as the simulator serves as a debugging tool thereafter. It is sometimes difficult to discover a nature of the captured error – a fault in the model, a property that doesn't satisfy or a wrong requirement. In this case the simulator shows the output trace that leads to the error and assist to discover the reason. Also, developers of the tool are very helpful in their feedback through the UPPAAL bug report system [1].

The case study has shown that UPPAAL is a suitable option for the verification of GAP, but might be a "bad" choice for verification of TCA-GAP due to incompatibilities mentioned above. And, in general, the model-checking for the tree-based aggregation protocols is a way to study and understand the behaviour of the system step by step for a small number of processes, but is not a scalable approach to verify due to the complexity of these protocols.

# Chapter 4

## Conclusion

A tree-based aggregation protocol computes an aggregation function in a decentralized way using an aggregation tree.

The thesis work considers this class of protocols and consists of two parts. The first part is the clarification of common concepts for the class of tree-based aggregation protocols including key requirements, related protocols and brief overview of existing optimization policies. The next part is a case study of Generic Aggregation Protocol (GAP). A part of the work is greatly aided with the use of automated reasoning tool – UPPAAL, a model checking tool for networks of timed automata. In the second part, the GAP protocol is examined for several requirements, including deadlock freedom, liveness and safety properties using UPPAAL.

As it was mentioned previously, a few aspects of the GAP protocol hasn't been considered in this work, such as the presence of node's failure and the local weight change. Different mechanisms have been implemented for the GAP model to overcome the problem with complexity and synchrony. However, no evaluation has been done related to their efficiency compared to another options for the solution of the same problems. In this way, the GAP model, presented in the previous chapter, might be improved. For example, a multicast implemented via manual traversal of the neighbourhood table can be replaced by conditional broadcast using the UPPAAL's primitive. Two models should be compared – whether the number of the messages send in case of broadcast consumes less memory than in the case of neighbourhood table traversal.

Using the experience of this work, one can try to model either GAP or other tree-based aggregation protocol using another model-checking tool. It is also possible to ensure the same behaviour for the GAP model with respect to the original protocol description using the theory of timed automata and bisimulation.

The case study of tree-based aggregation protocols can be continued in several directions. An experimental work can be done to study optimization policies for tree-based aggregation protocols. The comparison with the previously studied policies is possible as well.

Also, protocol verification is an area of the research interest. There are several works that can be used for the verification of the GAP protocol.

Dolev [12] studied time-optimal self-stabilizing dynamic protocols for a variety of tasks including topology update, routing and leader election. He gave both correctness and complexity proofs for the multiple BFS trees protocol and the counting protocol.

Another study done by Gartner and Pagnia [16] presented a method of combining a self-stabilizing algorithm with a hierarchical structure for construction of a self-stabilizing algorithms with improved stabilization time complexity. They proved that this method reduces stabilization time complexity to

a logarithmic one.

Fair combination of the self-stabilization protocol technique introduced by Dolev *et al.* [11] and Dolev [13] can be adapted for verification of the TCA-GAP protocol.

# Bibliography

- [1] *Uppaal's bug report system*.  
<http://bugsy.grid.aau.dk/cgi-bin/bugzilla/index.cgi>.
- [2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science (TCS)*, 126(2):183235, 1994.
- [3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
- [4] J. Bengtsson. Efficient symbolic state exploration of timed systems: Theory and implementation. Licentiate Thesis 2001-009, Department of Information Technology, Uppsala University, 2001.
- [5] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, volume 3098 of LNCS. Springer–Verlag, 2004.
- [6] D. Bosnacki and D. Dams. Integrating Real Time into Spin: A Prototype Implementation. In S. Budkowski, A. R. Cavalli, and E. Najm, editors, *FORTE*, volume 135 of *IFIP Conference Proceedings*, pages 423–438. Kluwer, 1998.
- [7] D. Bosnacki, D. Dams, and L. Holenderski. A heuristic for symmetry reductions with scalarsets. In J. Oliveira and P. Zave, editors, *FME 2001*, volume 2021 of LNCS, pages 518–533. Springer–Verlag, 2001.
- [8] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer–Verlag.
- [9] M. Dam and R. Stadler. A Generic Protocol for Network State Aggregation. In *Proc. Radiovetenskap och Kommunikation (RVK)*, Linköping, Sweden, June 2005.
- [10] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.
- [11] D. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

- [12] S. Dolev. Optimal time self stabilization in dynamic systems (*preliminary version*). In A. Schiper, editor, *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG93)*, volume 725 of *LNCS*, pages 160–173, Lausanne, Switzerland, 2729 September 1993. Springer–Verlag.
- [13] S. Dolev. *Self-Stabilization*. The MIT Press, Cambridge, MA, USA, 2000.
- [14] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
- [15] F. C. Gartner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Technical report, EPFL, 2003.
- [16] F. C. Gartner and H. Pagnia. Time-Efficient Self-Stabilizing Algorithms through Hierarchical Structures. In *6th Symposium on Self-Stabilizing Systems*, *LNCS*. Springer-Verlag, June 2003.
- [17] P. Hefti. A simple overlay platform for distributed monitoring. Master’s thesis, KTH, Royal Institute of Technology, Stockholm, April 2006.
- [18] M. Hendriks. Enhancing Uppaal by exploiting symmetry. Technical Report NIII-R0208, NIII, University of Nijmegen, October 2002.
- [19] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to UPPAAL. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS’03)*, number 2791 in *LNCS*, pages 46–59. Springer–Verlag, 2004.
- [20] G. J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [21] C. Ip and D. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, volume 725 of *LNCS*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, Journal version appeared in *Formal Methods in System Design*, 9(1/2):4175, 1996.
- [22] S. Keshav. Efficient and decentralized computation of approximate global state. *ACM Computer Communication Review*, January 2006.
- [23] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume III of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, Second edition, 1998.
- [24] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.
- [25] A. Leinwand and K. Fang. *Network Management. A Practical Perspective*. Addison-Wesley, second edition, 1996.
- [26] J. Liebeherr, M. Nahas, and W. Si. Application-Layer Multicasting With Delaunay Triangulation Overlays. *IEEE Journal on Selected Areas in Communications*, 20(8):1472–1488, October 2002.

- [27] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for ad hoc Sensor Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, USA, December 2002.
- [28] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 1992.
- [29] A. Montresor, M. Jelasity, and O. Babaoglu. Robust Aggregation Protocols for Large-Scale Overlay Networks. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, pages 19–28, Florence, Italy, June 2004. IEEE Computer Society.
- [30] P. Pan, E. Hahne, and H. Schulzrinne. BGRP: Sink-Tree-Based Aggregation for Inter-Domain Reservations. *Journal of Communications and Networks*, 2(2):157–167, June 2000. Special Issue on QoS in IP Networks.
- [31] G. Panno. Experimental evaluation of Tree-Based and Gossip-Based Aggregation Protocol. Master's thesis, KTH, Royal Institute of Technology, Stockholm, December 2005.
- [32] D. Peled. *Software Reliability Methods*. Texts in Computer Science. Springer, 2001.
- [33] A. G. Prieto and R. Stadler. Distributed real-time monitoring with accuracy objectives. In *IFIP Networking 2006*, May 2006.
- [34] G. Tel. *Introduction to Distributed Algorithms*. MIT Press, Cambridge, MA, USA, second edition, September 2000.
- [35] S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Proceedings of the Second International Workshop, TACAS'96*, volume 1055 of LNCS, pages 329–348. Springer-Verlag, 1996.
- [36] R. van Renesse. The importance of aggregation. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, number 2584 in LNCS, pages 87–92, Heidelberg, Germany, April 2003. Springer.
- [37] F. Wuhib. Robust Pattern for Decentralized Management. A Functional and Performance Evaluation. Master's thesis, KTH, Royal Institute of Technology, Stockholm, April 2005.
- [38] F. Wuhib, M. Dam, R. Stadler, and A. Clemm. Decentralized computation of threshold crossing alerts. In *16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2005, Barcelona, Spain, October 2005*, number 3775 in LNCS, pages 220–232, Oct. 2005.
- [39] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

# Appendix A

## Pseudocode

### A.1 GAP Pseudocode

Listing A.1 is the pseudocode of GAP.

```
(* Datatypes *)

(*message types*)
datatype mtype = {new, fail, update, weight, timeout};

datatype nodeId = "the node's identifier; natural, incl. 0 for root";

datatype From = "nodeId of the sender of a message";

datatype Weight = "local aggregate_weight";

datatype Level = "the level in BFS tree, natural incl. 0";

datatype Status = {self, parent, child, peer};

datatype Parent = "specifies parent id";

(*list of tuples*)
datatype Table = [(nodeId,Weight,Level,Status)] ;

(* auxillary functions *)

fun row(nodeId,Table)= "the entry in Table for nodeId";

fun self()= "nodeId of the requesting process";

(*update vector*)
fun updatevector(Table) =
  {update, self(),aggregate(),level(self()),parent()};

fun newentry(nodeId,Table) = Table::(nodeId,peer,0,0);
(*precondition nodeId is not in Table*)

fun removeentry(nodeId,Table) = "remove tuple of nodeId from Table";
```

```

proc updateentry(Table,From,Weight,Level,Parent) = {
(* receives update vector and updates the table *)
  if (Parent == self()) {Status = child }
  else
  if (row(From,Table).Status == child) {Status = peer } ;
  fi;
  row(From,Table).Status = Status;
  row(From,Table).Weight = Weight;
  row(From,Table).Level = Level;
};

fun level(nodeId)= "level of the node nodeId";

fun parent()= "if exists {the node id of entry labelled parent in table of
the current node} else not defined";

fun send(nodeId,vector)= {
  "sending {\tt update} message with 'vector' to the node with 'nodeId
'"};

fun broadcast(vector)=
  forall (Node=nodeId ∈ Table) {send(Node,vector)} ;

fun aggregate()= "aggregate weight over all entries in the table of the
current node";

(* main procedure *)

proc gap(Table) = { (* initialization *)
  init
  if "invoked as root" then
    Table = [(self(),parent,-1,0)] ; (*(virtualRoot(),parent,-1,0)*)
  fi;
  Table = [(self(),self,0,0)] ;
  "initialize (* operation *)
  failure detection/neighbour discovery,
  weight_subscription and
  timer services" ;
  Timeout = 0 ;
  NewNode = null;
  Vector = updatevector(Table) ;
  NewVector = Vector ;
  while true do
    receive
      (new,From) =>
        {newentry(From,Table) ;
        NewNode = From;}
      | (fail,From) =>
        removeentry(From,Table);
      | (update,From,Weight,Level,Parent) =>
        updateentry(Table,From,Weight,Level,Parent);
      | (weight,Weight) =>
        updateentry(Table,self(),Weight,level(self()),parent());
      | (timeout) => Timeout = 1 ;
  end while
}

```

```

    end ;
    restoreTableInvariant(Table) ; //see below
    NewVector = updatevector(Table) ;
    if NewNode!= null {
        send(NewNode,newVector) ;
        NewNode = null; }
    if NewVector != Vector && TimeOut {

        broadcast(newVector);
        TimeOut = 0 ;
        Vector=NewVector ; }

    fi;
od; } ; (* gap *)

proc restoreTableInvariant(Table) = {
    (* Conservative policy *)
    if Table has more than one entry
    then
        if Table has no parent
        then
            NewParent = nominate node with minimal level(Table) ;
            (*do not take into account level(self())*)
            (*if there are two entries of the min level ,
            pick one randomly as a parent*)
            row(NewParent,Table).Status = parent ;
        else if there is node with level less than level(parent(Table))
        then
            NewParent = nominate node with minimal level(Table) ;
            row(parent(Table),Table).Status = peer ;
            row(NewParent,Table).Status = parent;
        fi ;
        row(self(),Table).Level = row(parent(Table),Table).Level + 1
    fi ;
} (* restoreTableInvariant *)

```

*Listing A.1: Full GAP pseudocode*

## A.2 TCA-GAP Pseudocode

```

(* main procedure *)
proc tca-gap(Table, k1, k2) = { (* initialization *)
    Table = [self(),self,0,0,nil] ;
    TimeOut = 0 ;
    NewNode = null;
    "initialize failure detection and neighbor discovery services" ; (*
    operation *)
    Vector = updatevector(Table) ;
    newVector = Vector ;
    while true do
        receive
            (new,From) =>
                { newentry(From,Table) ;
                  NewNode = From;}
            | (fail,From) =>

```

```

    removeentry(From, Table)
| (update, From, Weight, Level, Parent, ThresholdList, Sign) =>
    updateTable(Table, From, Weight, Level, Parent, ThresholdList, Sign)
| (weight, Weight) =>
    updateentry(self(), Weight, level(self()), parent(), ThresholdList, Sign)
| (timeout) => Timeout = 1
end ;
if (thresholdCrossed()) {
    (* check whether aggregate crossed threshold *)
    sendTCAtoMgmtStation();
    switchMode();
    (* change the mode, use the other threshold/sign *)
}
restoreTableInvariant(Table) ; //see below
NewVector = updatevector(Table) ;
if NewNode != null {
    send(NewNode, newVector) ;
    NewNode = null; }
if NewVector != Vector && Timeout {
if "only aggregate changed" {
    send(parent(), newVector) }
else {
    broadcast(newVector) } ;
Timeout = 0 ;
Vector=NewVector }
od } ; (* tca-gap *)

proc restoreTableInvariant(Table) = {
(* see restoreTableInvariant procedure of GAP *)
"check that parent labelling is correct" ;
if localthreshold == 0 (* node should be active *)
    or (sign * "local aggregate" > sign * localthreshold *  $k_1^{sign}$ ) {
        "set thresholds of children to 0";
        exit
    } ;
if  $\sum(\text{thresholds of children}) > 0$  {
    (* node is passive *)
    Needed = sign * ( $\sum(\text{"thresholds of children"}) - \text{localthreshold}$ ) ;
if (Needed > 0) {
    (* means parent reduced threshold *)
    "Reduce/increase threshold of passive children by 'Needed'" } ;
if sign *  $\sum(\text{"aggregate active children"}) >$ 
    sign *  $\sum(\text{"threshold active children"})$  {
    (* risk of local threshold passing *)
    k = "an active node with largest sign * (aggr - threshold)" ;
    "Reduce/increase threshold of passive children by
        sign * (aggrk / ( $k_1^{sign}$ ) - thresholdk)" ;
    thresholdk = aggrk / ( $k_1^{sign}$ ) } }
else {
    if (sign * "local aggregate" < sign * localthreshold *  $k_2^{sign}$ ) {
        for node i ∈ "children" do
            (* switch to passive *)
            thresholdi = localthreshold * weighti / "local aggregate" }}
} (* restoreTableInvariant *)

```

Listing A.2: TCA-GAP pseudocode

# Appendix B

## GAP code in UPPAAL

### B.1 Full model

#### B.1.1 System global code

```
//Insert declarations of global clocks, variables, constants and channels.
const int n = 6; // maximum number of nodes
const int max = 10; // range upper limit

typedef int [0,max] byte;
typedef int [-1,max] nbyte;
typedef int [0,n] nty;
typedef int [0,1] bit;
typedef int [0,n-1] id_t;

byte ms_id, ms_wei;
nbyte ms_lev, ms_par; // shared message fields

const int to = 15; //timeout interval

const int self = 1;
const int parent = 4;
const int peer = 3;
const int child = 2;

nty cnt=n;

bool esc[n+1];

chan update[n+1]; // message channels
chan new[n+1];
chan fail[n+1];
chan weight[n+1];

chan bupdate[n+1]; // message channels
chan bnew[n+1];
chan bfail[n+1];
chan bweight[n+1];

clock GTime;
```

```

bool go(){
  if (cnt==0) {return true;} else {return false;}
}

```

*Listing B.1: System global entities declaration code*

## B.1.2 Node template

```
//Insert local declarations of clocks, variables and constants.
```

```

nbyte table[n+1][3];

bit TimeOut=0;
nbyte Vector[4]={0,0,0,0}, NewVector[4]={0,0,0,0};
byte newn;
bool bc = false;
bool newM = false;

nbyte plevel;
byte p=0;
int[1,n+1] k=1;
nty no=1;

clock t;

bool one=true;
bool hasP=false;

```

```
//Insert local declarations of functions.
```

```
//node starts
```

```

void startInit(){
  if (nid!=0){ Vector[1]=-1; }
  if (nid==0)
  {
    table[0][1]=-1;
    table[0][0]=parent;
    table[0][2]=0;
    hasP=true;
    one=false;
    Vector[1]=nid;
    p=0;
    plevel=-1;
  }
  Vector[0]=nid;
  Vector[2]=1;
  Vector[3]=w;
  table[nid+1][1]=1;
  table[nid+1][0]=self;
  table[nid+1][2]=w;
  TimeOut=0;
  t=0;
  NewVector[0]=Vector[0];
  NewVector[1]=Vector[1];
  NewVector[2]=Vector[2];

```

```

    NewVector [3]=Vector [3];
}

void count(){
    cnt--;
}

//new message processing
void newadd()
{
    t=0;
    k=1;
    if (table[ms_id+1][0]==0){ //check if this entry doesn't exist
        table[ms_id+1][0]=peer;
        table[ms_id+1][1]=0;
        table[ms_id+1][2]=0;
        one=false;
        no++;
        newM = true;
        newn=ms_id;
    }
    ms_id=0;
}

void failrem(){
    k=1;
    if (table[ms_id+1][0]!=0){
        no--;
        table[ms_id+1][0]=0;
        table[ms_id+1][1]=0;
        table[ms_id+1][2]=0;
        if (nid!=0 && no==1) {one=true;}
        else if (nid==0 && no==1) {one=false;}//it has a vr
        else if (p==ms_id) {p=0; hasP=false; plevel=0;}
    }
    ms_id=0;
}

}

//update message processing - converting parentId into status
void updateadd(){
    k=1;
    if (table[ms_id+1][0]==0){no++; one=false;}
//statusFix part
    if (ms_par==nid) {table[ms_id+1][0]=child;}
    else if ((ms_par!=nid) && (table[ms_id+1][0]==0))
        {table[ms_id+1][0]=peer; one=false;}
    else if ((table[ms_id+1][0]==child) && (ms_par!=nid))
        {table[ms_id+1][0]=peer;}
//add this information to the table
    table[ms_id+1][1]=ms_lev;
    table[ms_id+1][2]=ms_wei;
//check the change in parent information
    if (p==ms_id){
        //if status changed, search for another parent
        if (table[ms_id+1][0]!=parent)

```

```

        {plevel=0; hasP=false; p=0;}
//else, just update the level
else {plevel=ms_lev;}
}
}

//restoreTableInvariant procedure
void resTabInv(){
int[1,n+1] i;
byte pold;
if (!one){
    if (!hasP){
        while ((table[i][0]==0) || (table[i][0]==self) && (i<n+1))
            {i++;}
        if ((table[i][0]!=0) && (table[i][0]!=self)){
            plevel=table[i][1];
            p=i-1;
            table[i][0]=parent;
            hasP=true;
        }
    }
    if (hasP){
        pold=p;
        for (i=1;i<=n;i++){
            if ((table[i][0]!=0) && (table[i][0]!=self)
                && (table[i][1]<plevel)){
                table[pold+1][0]=peer;
                plevel=table[i][1];
                p=i-1;
                table[i][0]=parent;
            }
        }
        table[nid+1][1]=plevel+1;
    }
}
}

//generating of UpdateVector
void genUpVector(){
int[1,n+1] i;
bit aggr;
for (i=1;i<=n;i++){
    if ((table[i][0]==child) || (table[i][0]==self)){
        aggr = (aggr + 1) % 2;
    }
}
NewVector[0]=nid;
NewVector[1]=p;
NewVector[2]=table[nid+1][1];
NewVector[3]=aggr;
}

//invariant for restoreTableInvariant
bool invTable(){
bool result=false;
if (((table[nid+1][1]==(plevel+1))

```

```

        && (table[nid+1][0]==self)) || (one)){
            result=true;
        }
        return result;
    }

bool newVectorEq(){
    bool result=false;
    if ((NewVector[0]!=Vector[0]) ||
        (NewVector[1]!=Vector[1]) ||
        (NewVector[2]!=Vector[2]) ||
        (NewVector[3]!=Vector[3])){
        result=true;
    }
    return result;
}

void resetSharVar(){
    ms_id=0;
    ms_wei=0;
    ms_lev=0;
    ms_par=0;
}

void arraysMerge(){
    Vector[0]=NewVector[0];
    Vector[1]=NewVector[1];
    Vector[2]=NewVector[2];
    Vector[3]=NewVector[3];
}

void prepUpVec(){
    ms_wei=NewVector[3];
    ms_lev=NewVector[2];
    ms_par=NewVector[1];
    ms_id=NewVector[0];
}

void newRes(){
    newn=0;
    newM=false;
}

```

*Listing B.2: UPPAAL Node's code*

### B.1.3 Buffer template

```

//Insert local declarations of clocks, variables and constants.

nbyte bin2, bin3; //local variable, just for store received information
byte bin, bin4;

void receiveUp(){
    bin=ms_id;
    bin2=ms_par;
}

```

```

    bin3=ms_lev;
    bin4=ms_wei;
}

void sendUp(){
    ms_id=bin;
    ms_par=bin2;
    ms_lev=bin3;
    ms_wei=bin4;
}

void resetSharVar(){
    ms_id=0;
    ms_wei=0;
    ms_lev=0;
    ms_par=0;
}

void resetVar(){
    bin=0;
    bin2=0;
    bin3=0;
    bin4=0;
}

```

*Listing B.3: UPPAAL Buffer's code*

## B.1.4 System declaration

```

//Insert process assignments.
id_t ar;

node(const id_t id)=Node(id,ar,1,0);

Discovery0:=Discovery(0,1);
Discovery1:=Discovery(1,3);
Discovery2:=Discovery(0,2);
Discovery3:=Discovery(1,2);
Discovery4:=Discovery(1,4);
Discovery5:=Discovery(3,4);
Discovery6:=Discovery(4,5);

buffer(const id_t id)=Buffer(id,ar);

//Edit system definition.
system node, buffer, Discovery1, Discovery2, Discovery3,
    Discovery4, Discovery5, Discovery6;

```

*Listing B.4: System definition*

## B.2 Simplified model

### B.2.1 System global code

```

//Insert declarations of global clocks, variables, constants and channels.
const int n = 6; // maximum number of nodes
const int max = 10; // range upper limit

typedef int [0,max] byte;
typedef int [-1,max] nbyte;
typedef int [0,n] nty;
typedef int [0,1] bit;
typedef int [0,n-1] id_t;

byte ms_id, ms_wei;
nbyte ms_lev, ms_par; // shared message fields

const int to = 15; //timeout interval

const int self = 1;
const int parent = 4;
const int peer = 3;
const int child = 2;

nty cnt=n;

bool esc[n+1];//

chan update[n+1]; // message channels
chan new[n+1];

chan bupdate[n+1]; // message channels
chan bnew[n+1];

clock GTime;

bool go(){
  if (cnt==0) {return true;} else {return false;}
}

```

*Listing B.5: System global entities declaration code*

## B.2.2 Node template

```

//Insert local declarations of clocks, variables and constants.

nbyte table[n+1][3];

bit TimeOut=0;
nbyte Vector[4]={0,0,0,0}, NewVector[4]={0,0,0,0};
byte newn;
bool bc = false;
bool newM = false;

nbyte plevel;
byte p=0;
int [1,n+1] k=1;

clock t;

```

```

bool one=true;
bool hasP=false;

//Insert local declarations of functions.
//node starts
void startInit(){
    if (nid!=0){ Vector[1]=-1; }
    if (nid==0)
        {
            table[0][1]=-1;
            table[0][0]=parent;
            table[0][2]=0;
            hasP=true;
            one=false;
            Vector[1]=nid;
            p=0;
            plevel=-1;
        }
    Vector[0]=nid;
    Vector[2]=1;
    Vector[3]=w;
    table[nid+1][1]=1;
    table[nid+1][0]=self;
    table[nid+1][2]=w;
    TimeOut=0;
    t=0;
    NewVector[0]=Vector[0];
    NewVector[1]=Vector[1];
    NewVector[2]=Vector[2];
    NewVector[3]=Vector[3];
}

void count(){
    cnt--;
}

//new message processing
void newadd()
{
    t=0;
    k=1;
    if (table[ms_id+1][0]==0){ //check if this entry doesn't exist
        table[ms_id+1][0]=peer;
        table[ms_id+1][1]=0;
        table[ms_id+1][2]=0;
        one=false;
        newM = true;
        newn=ms_id;
    }
    ms_id=0;
}

//update message processing - converting parentId into status
void updateadd(){
    k=1;

```

```

    if (table[ms_id+1][0]==0){one=false;}
//statusFix part
    if (ms_par==nid) {table[ms_id+1][0]=child;}
    else if ((ms_par!=nid) && (table[ms_id+1][0]==0)){table[ms_id+1][0]=peer
; one=false;}
    else if ((table[ms_id+1][0]==child) && (ms_par!=nid)){table[ms_id+1][0]=
peer;}
//add this information to the table
    table[ms_id+1][1]=ms_lev;
    table[ms_id+1][2]=ms_wei;
//check the change in parent information
    if (p==ms_id){
        //if status changed, search for another parent
        if (table[ms_id+1][0]!=parent) {plevel=0; hasP=false; p=0;}
        //else, just update the level
        else {plevel=ms_lev;}
    }
}

//restoreTableInvariant procedure
void resTabInv(){
int[1,n+1] i;
byte pold;
if (!one){
    if (!hasP){
        while ((table[i][0]==0) || (table[i][0]==self) && (i<n+1)) {i++;}
        if ((table[i][0]!=0) && (table[i][0]!=self)){
            plevel=table[i][1];
            p=i-1;
            table[i][0]=parent;
            hasP=true;
        }
    }
    if (hasP){
        pold=p;
        for (i=1;i<=n;i++){
            if ((table[i][0]!=0) && (table[i][0]!=self) && (table[i][1]<
plevel)){
                table[pold+1][0]=peer;
                plevel=table[i][1];
                p=i-1;
                table[i][0]=parent;
            }
        }
        table[nid+1][1]=plevel+1;
    }
}
}

//generating of UpdateVector
void genUpVector(){
int[1,n+1] i;
bit aggr;
for (i=1;i<=n;i++){
    if ((table[i][0]==child) || (table[i][0]==self)){
        aggr = (aggr + 1) % 2;
    }
}
}

```

```

    }
  }
  NewVector[0]=nid;
  NewVector[1]=p;
  NewVector[2]=table[nid+1][1];
  NewVector[3]=aggr;
}

//invariant for restoreTableInvariant
bool invTable(){
  bool result=false;
  if (((table[nid+1][1]==(plevel+1)) && (table[nid+1][0]==self)) || (one))
  {
    result=true;
  }
  return result;
}

bool newVectorEq(){
  bool result=false;
  if ((NewVector[0]!=Vector[0]) || (NewVector[1]!=Vector[1]) ||
      (NewVector[2]!=Vector[2]) || (NewVector[3]!=Vector[3])){
    result=true;
  }
  return result;
}

void resetSharVar(){
  ms_id=0;
  ms_wei=0;
  ms_lev=0;
  ms_par=0;
}

void arraysMerge(){
  Vector[0]=NewVector[0];
  Vector[1]=NewVector[1];
  Vector[2]=NewVector[2];
  Vector[3]=NewVector[3];
}

void prepUpVec(){
  ms_wei=NewVector[3];
  ms_lev=NewVector[2];
  ms_par=NewVector[1];
  ms_id=NewVector[0];
}

void newRes(){
  newn=0;
  newM=false;
}

```

*Listing B.6: UPPAAL Node's code*

### B.2.3 Buffer template

```
//Insert local declarations of clocks, variables and constants.

nbyte bin2, bin3; //local variable, just for store received information
byte bin, bin4;

void receiveUp(){
    bin=ms_id;
    bin2=ms_par;
    bin3=ms_lev;
    bin4=ms_wei;
}

void sendUp(){
    ms_id=bin;
    ms_par=bin2;
    ms_lev=bin3;
    ms_wei=bin4;
}

void resetSharVar(){
    ms_id=0;
    ms_wei=0;
    ms_lev=0;
    ms_par=0;
}

void resetVar(){
    bin=0;
    bin2=0;
    bin3=0;
    bin4=0;
}
```

*Listing B.7: UPPAAL Buffer's code*

### B.2.4 System declaration

```
//Insert process assignments.
id_t ar;

node(const id_t id)=Node(id,ar,1,0);

Discovery0:=Discovery(0,1);
Discovery1:=Discovery(1,3);
Discovery2:=Discovery(0,2);
Discovery3:=Discovery(1,2);
Discovery4:=Discovery(1,4);
Discovery5:=Discovery(3,4);
Discovery6:=Discovery(4,5);

buffer(const id_t id)=Buffer(id,ar);

//Edit system definition.
```

---

```
system node, buffer, Discovery1, Discovery2, Discovery3,  
        Discovery4, Discovery5, Discovery6;
```

*Listing B.8: System definition*