



Dubravka Ilić | Elena Troubitsyna

A Formal Model-Driven Approach to Requirements Engineering

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 667, February 2005



A Formal Model-Driven Approach to Requirements Engineering

Dubravka Ilić

Åbo Akademi University, Department of Computer Science

Elena Troubitsyna

Åbo Akademi University, Department of Computer Science

TUCS Technical Report
No 667, February 2005

Abstract

Model Driven Architecture (MDA) gains increasing acceptance in software engineering community. MDA promotes system development by gradual transformation of system models expressed in Unified Modelling Language (UML). UML modelling facilitates better understanding of system requirements, but it is yet insufficient for guaranteeing overall correctness of the final product. In this paper we propose an approach to formalizing model-driven development in the B Method. The B Method is a top-down approach to the development of systems correct by construction. We show how the proposed approach facilitates structuring complex system requirements, requirements changes and traceability, integration of emergent requirements and navigation through the overall design space. To validate the proposed approach we conduct a case study – development of Ad hoc On-Demand Distant Vector routing protocol.

Keywords: MDA, UML, B Method, refinement, requirements traceability and change

TUCS Laboratory
Distributed Systems Design Laboratory

1. Introduction

UML [10] has become de-facto a standard modeling technique for visualizing, specifying and documenting software systems. UML facilitates requirements analysis and specification. UML helps to visualize and express the system structure and behaviour at different levels of abstraction. Transformations of system models expressed in UML are in the basis of Model Driven Architecture (MDA) [8]. MDA is a methodology promoting top-down system development by transformation of abstract, platform independent models to platform specific models. Transformation steps, called refinements, aim at gradual introduction of requirements into system model. Correctness of both, models and transformations, is crucial for the quality of the final product. In this paper we propose a formal model-driven approach to requirements engineering with UML.

To verify consistency of our UML models we demonstrate how to translate them into a formal specification in the B Method [1, 11]. We extend the previous work on translating state and class diagrams into B by defining how in addition activity diagrams can be translated and integrated into the process of translating class and state diagram. While translating state and class diagrams we obtain a formal specification which lacks the details of implementation of class methods. After translating the corresponding activity diagrams we arrive at the complete formal specification.

The top-down development methodology of the B method, called stepwise refinement, coincides with the idea of MDA development by model transformations and hence, their combination is natural. We show how MDA in combination with formal refinement in B can result in an implementation of a system correct by construction. Since requirements changes and evolution are intrinsic part of software development process we show how they can be handled in the process of UML model transformation and corresponding formal refinement in B. We propose UML patterns for integrating arising requirements into the system models via model transformation. Establishing refinement between formal specifications representing system models before and after the transformation allows us to ensure correctness of model transformation.

Since the derivation of technology independent patterns is impractical, our approach is aiming at developing specification and development patterns generic to ad-hoc mobile networks. We create UML-based templates for modeling ad-hoc networks and corresponding patterns for specifying them in B. Moreover, we propose a generic development process based on transformations of UML models. By establishing refinement between the corresponding B models we verify the correctness of our development.

Our approach allows us to understand and structure complex system requirements and to reason about correctness of the system under construction.

To validate the proposed approach we conducted a case study – model-driven development of routing protocol for ad hoc networks, Ad hoc On-Demand Distance Vector protocol [9].

1. Modeling system requirements in UML

2.1. Requirements modeling with UML

A process of software development is a process of modeling a real world problem described as a set of requirements, transforming it into a number of refined models, eventually ending with executable code. Usually use case diagrams serve as a starting point for requirements modeling. From the functional requirements described by use cases we create a system structure, traditionally rendered by class diagrams.

To establish conformance to the requirements we should ensure that on the basis of created static model – classes with their associations, attributes and methods – the behaviour described by use case model can be provided. The dynamic aspect of behaviour is usually captured via state diagrams. The state space of a class is a Cartesian product of the types of class attributes depicted at the class diagram. The transitions from state to state correspond to the invocation of corresponding methods. The initialization of class attributes brings the class to its initial state.

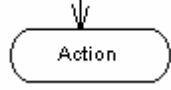
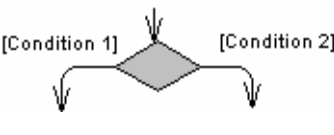
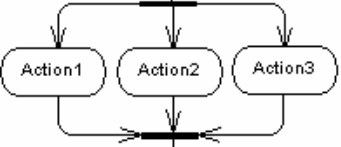
State diagrams can model the requirements on different level – from dynamic behaviour of a certain class to overall system behaviour. In the later case the states are formed from the attributes of all classes. The state transitions correspond to methods of certain classes.

While state diagram describes how an invocation of certain methods affects the state of the class it leaves the detailed specification of the computation implemented by these methods aside. The activity diagrams help us to unfold these details by describing how methods are actually implemented. To construct activity diagram we distinguish between basic and complex methods. The basic methods are the ones whose invocation changes or access a single attribute of the class. The complex methods describe elaborated computation and often are composed of basic methods. In our activity diagrams an activity – an execution flow step – corresponds to an execution of a basic method. Sequential dependencies between the execution steps are expressed as transitions.

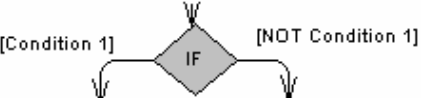

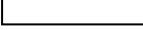
In this paper we demonstrate how UML modeling can assist us in handling requirements changes in a structured systematic way. While the connection between class and state diagrams in the process of requirements change has been studied before [6], the affect of it on the connection between the class, state and activity diagram has not been sufficiently

explored yet. Meanwhile, changing requirements usually have a profound impact on the way the methods are implemented. Next we will describe how to modify the activity diagram to facilitate the requirements changes.

Usually an activity diagram has the following three standard elements:

1)		basic method
2)		sequential branch with guard conditions on multiple exit arrows
3)		parallel execution of basic methods

Below we propose a modification of activity diagram to make the meaning of several elements more precise. Namely, we introduce the following variations:

4)		alternative branch with alternative guards on exit arrows
5)		nondeterministic branch with multiple exit arrows without guards
6)		block

The first element is essentially a specialization of the sequential branch. It renders the alternative choice. The second element is a specialization of the sequential branch to model non-determinism, i.e., the choice with overlapping conditions. The third element is a block. It models “folded” activity diagram and serves as a reference. An introduction of block improves scalability of the activity diagrams.

To illustrate modeling the requirements with class, state and activity diagram next we will present a model of route manipulation in Ad hoc On-Demand Distant Vector routing protocol (AODV).

2.2. Example of requirements modeling

AODV has been proposed for ad-hoc mobile wireless networks [9]. Essentially the protocol describes the communication, i.e., sending and

receiving information (called packets), between the nodes of a network. Ad-hoc networking is a rather new and hence not thoroughly explored technology. The protocol is a complex and changes constantly as the technology evolves. Therefore it offers us a good test-case for exploring requirements change. We omit a detailed description of the entire protocol and present small excerpts from it. Since the topology of an ad-hoc network is volatile, sending data from one node to another requires establishing a route between these nodes. Each node in the network is uniquely identified. It stores and updates a routing table containing routes already found from this node, called source, to the other nodes, called destinations, in the network. When the route is requested at the first time, node inserts it into its routing table with the status unknown. When the route is found it is marked as valid which means it can be used for sending packets. However, it can also become invalid when network topology changes. When node wishes to send a packet to the destination marked in the route, it sends the packet to the node which is listed as a next node from the observed source node in the route. In this way, packets are propagated toward destination nodes. Packets are buffered. Buffering allows the node to not process packets with the same identifier more than once.

We list and label a subset of the described requirements which we model in this paper:

- [1] Each node has a unique ID
- [2] Each node can send packets it has received
- [3] Each node can receive packets
- [4] Each packet has its own unique ID
- [5] Each packet has its source and a destination
- [6] Each node maintains routs toward other nodes
- [7] Each node has its status; it is valid if the route can be used for sending packets, invalid if for various reasons this can't be done or unknown if it is not completely established
- [8] Each node is buffering received packets
- [9] If a node has a route toward the destination in its routing table, it sends the packet to the next node along this route and either just waits for other packets to be received or at the same time also buffers the sending packet.

Next, we show how each of listed requirements can be modeled in UML. Figure 1 gives an excerpt from the AODV modeled in UML. The complete activity diagram of the method Send can be found in the Appendix (Figure A1).

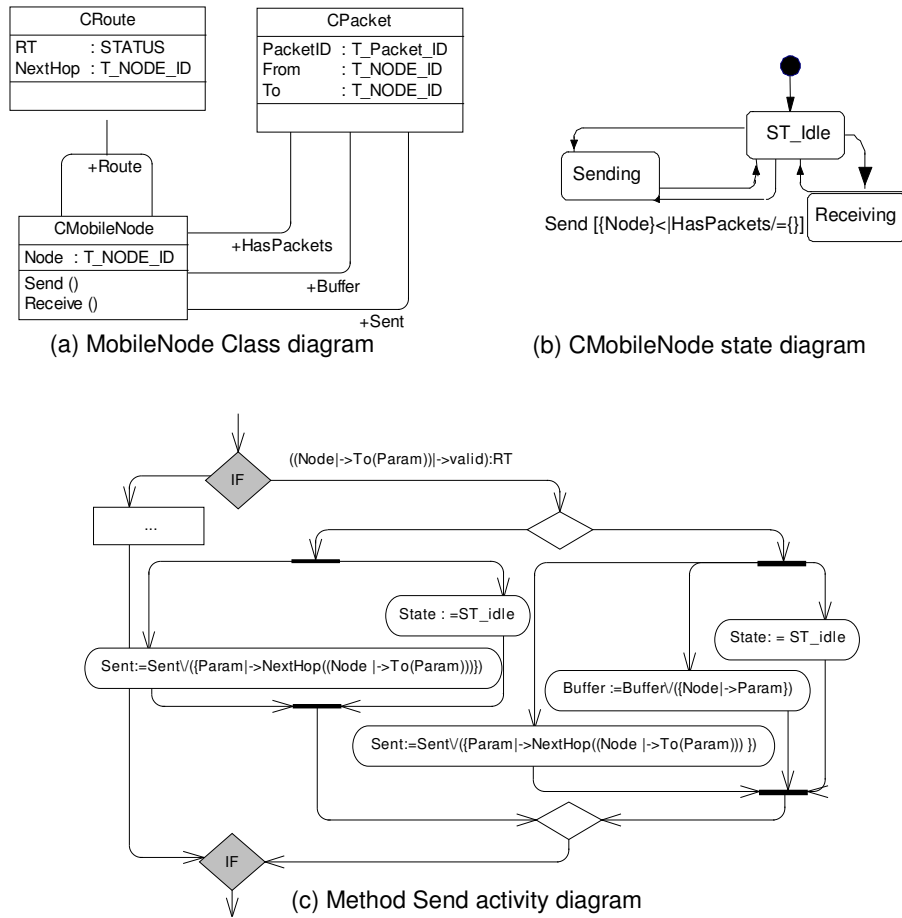


Figure 1. Excerpt from the AODV UML model

Observe that activity diagrams allow us to capture a significant part of the requirements, which we could not model with the class and state diagrams, as illustrated in Table 1.

Table 1. System requirements – abstract level

<i>Label</i>	<i>Description of requirement modelling</i>	<i>Used UML diagrams</i>
R1	Static feature of the node representing its unique identifier – Node – modeled as an attribute of the identified CMobileNode class.	Class diagram (Figure 1(a))
R2	Method Send of the CMobileNode class and at the same time association named Sent between classes CMobileNode and CPacket. R2 is modeled also as a state Sending and condition on a transition between states Idle and Sending - when some packet shows up in node's receiving channel HasPackets, node goes from Idle state to Sending state following the transition which invokes the method Send.	Class diagram CMobileNode State diagram (Figure 1(b))

R3	Method Receive of the CMobileNode class, and at the same time association HasPackets between classes CMobileNode and CPacket. R3 is modeled also as a state Receiving.	Class diagram CMobileNode State diagram (Figure 1(b))
R4	Class CPacket has its own ID attribute – PacketID.	Class diagram (Figure 1(a))
R5	Attributes of the CPacket class: From and To modeling the packet source and destination node.	Class diagram (Figure 1(a))
R6	The association Route together with the aggregated class CRoute models the node routing table.	Class diagram (Figure 1(a))
R7	NextHop is the attribute of the aggregated class CRoute modeling next node toward the destination node along the route; RT is the attribute of the aggregated class CRoute modeling its status.	Class diagram (Figure 1(a))
R8	Association relationship Buffer between classes CMobileNode and CPacket.	Class diagram (Figure 1(a))
R9	Body of the method Send.	Method Send activity diagram (Figure 1(c))

Next, we demonstrate our approach to verifying the consistency of our UML modeling.

3. From UML models to formal specifications

3.1. Formal system modeling in the B Method

To ensure consistency we supplement our UML models with formal specifications. In this paper we have chosen the B Method as our formal modeling framework. The B Method is an approach for the industrial development of correct software. The method has been successfully used in the development of several complex real-life applications [7]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [3], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

In B a specification is represented by a module or a set of modules, called Abstract Machines. The common pseudo-programming notation, called Abstract Machine Notation (AMN), is used to construct and formally verify them. An abstract machine encapsulates a state and operations of the specification and has the following general form:

MACHINE	name
SETS	Set
VARIABLES	v
INITIALISATION	Init
INVARIANT	I
OPERATIONS	Op

Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialized in the INITIALISATION clause. The variables in B are strongly typed by constraining predicates of INVARIANT clause. The constraining predicates are conjoint by conjunction (denoted as &). All types in B are represented by non-empty sets and hence set membership (denoted as :) expresses typing constraint for a variable, e.g., x:TYPE. Local types can be introduced by enumerating the elements of the type, e.g., TYPE = {element1, element2,...} in the SETS clause. The operations of the machine are defined in OPERATIONS clause. The operations are atomic meaning that, once an operation is chosen, its execution will run until completion without interference.

In this paper we adopt event-based approach to system modeling [2]. The events are specified as the guarded operations SELECT cond THEN body END. Here cond is a state predicate, and body is a B statement describing how state variables are affected by the operation. If cond is satisfied, the behaviour of the guarded operation corresponds to the execution of its body. If cond is false at the current state then the operation is disabled, i.e., cannot be executed. Event-based modeling is especially suitable for describing reactive systems. Then SELECT operation describes the reaction of the system when particular event occurs.

In this paper we use the following B statements to describe the computation in operations:

Statement	Informal meaning
X := e	Assignment
X, y := e1, e2	Multiple assignment
IF P THEN S1 ELSE S2 END	If P is true then execute S1, otherwise S2
S1 ; S2	Sequential composition
S1 S2	Parallel execution of S1 and S2
X :: T	Nondeterministic assignment – assigns variable x arbitrary value from given set T
ANY x WHERE Q THEN S END	Nondeterministic block – introduces new local variable x according to the predicate Q which is then used in S
CHOICE S OR T OR ... OR U END	Nondeterministic choice – one of the statements S, T...U is arbitrarily chosen for execution

B also provides structuring mechanisms which enable machines to be expressed as combinations of other machines. Here we use EXTENDS clause. When machine M1 extends machine M2, written as EXTENDS M2 in the definition of M1, it means that M1 includes M2 and promotes all of the operations of M2, i.e., it provides all of the facilities provided by M2, with some further operations of its own.

Constructing a formal specification from a requirements description given in a natural language is often cumbersome. In our approach the formal specifications are obtained by translating UML models. Next we demonstrate our approach to doing this.

3.2. Translating UML diagrams into B

In this paper we further extend an approach to translating class and state diagrams into B described in [12]. We start from a brief revision of the existing translation mechanism and then present our extension.

Classes in UML correspond to machines in B. In the process of translating a class into a machine we transform attributes into the variables and the methods into the operations. Initial values of the attributes become initial values of variables in the initialization clause. The tagged values of the class describing the invariant are translated into the machine invariant. The state diagram provides a basis for constructing operations of the machine. The names of the states form the set of values of the variable modeling state. The names of operations correspond to the names of the transitions. The guard of a transition forms the guard of the corresponding operation. For instance, as a result of translating class and state diagram presented in Figure 1(a) and 1(b), the B specification shown on Figure 2 can be obtained.

Observe that unidirectional associations between classes are modeled as relations between types of variables representing identifiers of the classes. In the B specification resulting from translating class and state diagram we have captured only the requirements. The body of the operation is still unspecified. Next we show how the translation of an activity diagram complements the body of the corresponding B operation.

Since B can express not only sequential but also parallel processes and their synchronization it is suitable for representing activity diagram formally.

```

MACHINE      CMobileNode
EXTENDS      CPacket, CRoute / Additional machines – see Appendix (Figure A2 b) and c))
SEES         Global / Definition of types – see Appendix (Figure A2 a))
VARIABLES
  Node, / attribute of CMobileNode class
  Route, / association between CMobileNode and CRoute classes
  HasPackets, / association between CMobileNode and CPacket classes
  Sent, Buffer, / association between CMobileNode and CPacket classes
  CMobileNode_State, / automatically generated variable for modeling the machine
                    state (name corresponds to the machine name)

INVARIANT
  Node : T_NODE_ID &
  Route <: T_NODE_ID <-> T_NODE_ID &
  HasPackets : T_NODE_ID <-> PacketID &
  CMobileNode_State : STATES &
  Sent : PacketID<-> T_NODE_ID &
  Buffer : T_NODE_ID <-> PacketID &

INITIALISATION

  Node:=IP_address || / initially Node gets a unique IP_address which is a constant
  Route:={} || / initially Node doesn't have any established routes
  HasPackets:={} || / initially Node doesn't have any sent, received or buffered
                  packets

  Sent:={} ||
  Buffer:={} ||
  CMobileNode_State:=ST_Idle || / initially Node is in the state Idle

OPERATIONS
  Send= / the operation is executed when the Node is in the state
        Sending it has some packets to send

  SELECT
    CMobileNode_State=Sending & {Node}<|HasPackets/={}
  THEN

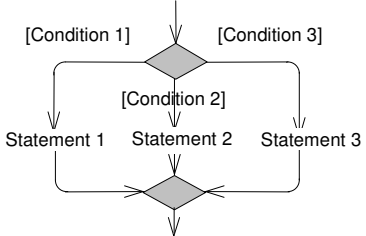
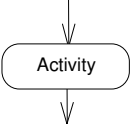
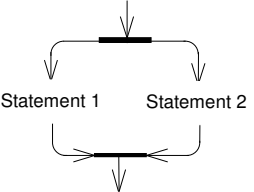
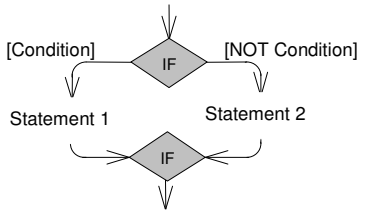
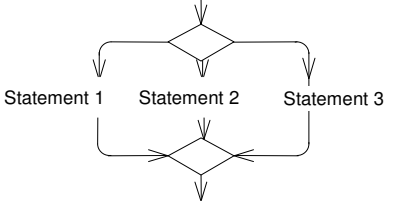
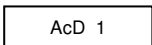
  END

  ...

```

Figure 2. Excerpt from the AODV class and state diagram translation into B

We define a mapping as a set of rules given below:

	UML element	Corresponding B notation
Rule 1	Activity diagram NAME	Body of operation NAME
Rule 2	Sequential branch 	IF Condition 1 THEN Statement 1 ELSIF Condition 2 THEN Statement 2 ELSIF Condition 3 THEN Statement 3 ELSE skip END
Rule 3	Basic method 	Assignment
Rule 4	Parallel execution 	Statement 1 Statement 2
Rule 5	Alternative branch 	IF Condition THEN Statement 1 ELSE Statement 2
Rule 6	Nondeterministic branch 	CHOICE Statement 1 OR Statement 2 OR Statement 3 END
Rule 7	Block  * There exist an activity diagram AcD_1	Corresponds to the translation of the contained activity diagram (see Rule 1).

To illustrate the translation of the activity diagrams into B in Figure 3 we present the results of translating the activity diagram given in Figure 1 (c). The translation of the complete activity diagram for the method Send can be found in Appendix (Figure A3).

```

MACHINE CMobileNode'
<specification of machine's sets, variables, invariants and initialization>
OPERATIONS
Send =
SELECT CMobileNode_State=Sending & { Node } <| HasPackets /= {}
  THEN
    <specification of the operation body>
    IF
      ( ( Node | -> To ( Param ) ) | -> valid ) : RT
      THEN
        Sent := Sent  $\vee$  ( { Param | -> NextHop ( ( Node | -> To ( Param ) ) ) } ) ||
        CHOICE
          CMobileNode_State:=ST_Idle
        OR
          Buffer := Buffer  $\vee$  ( { Node | -> Param } ) ||
        END
      ELSE
        <specification of the operation body -- contd>
    END
  END
END

```

/ If the route from the observed node to desired destination To in the routing table RT is valid

/ Then

/ Packet is sent to the next node along that route and node either becomes idle

/ or the packet is buffered and node goes to idle

Figure 3. Excerpt from the AODV activity diagram translation into B

Observe that translation of the activity diagram as represented by CMobileNode' in Figure 3 has allowed us to complete the formal specification of CMobileNode. Indeed, the translation has supplemented the body of the operation Send constructed as described by the corresponding activity diagram.

Currently the tool U2B supports the automatic translation of the class and state diagrams into B [13].

To achieve a completely automatic translation some restrictions on UML modeling should be imposed, e.g. logical conditions should be described in B notations. By defining logical conditions in terms of Object Constraint Language (OCL) and then automatically translating it into B, this restriction can be removed.

In this section we have demonstrated how to supplement UML modeling with formal specification which allowed us to ensure consistency of our UML models. We have demonstrated so called horizontal consistency of UML models. However, our specification is still on an abstract level and needs to be refined. In the following section we show how MDA in combination with formal refinement in B will lead us towards a correct system implementation.

4. B-supported model transformations in MDA

MDA has appeared as a result of recognizing that reasoning about software on the code level is unfeasible. MDA stresses an importance of abstraction - as a main mechanism for coping with complexity - and stepwise refinement - as a technique for transforming abstract models into an implementation on a desired platform. In the previous section we demonstrated how class, state and activity diagrams can capture different views on system behaviour and then shown how to ensure the consistency of these views by formal verification. In this section we will demonstrate how these models are transformed in the process of requirements evolution and change.

Refinement is a process of gradual incorporation of requirements and implementation details into the system specification. While incorporating new requirements we should ensure that the observable behaviour of the system is preserved, i.e., to guarantee an adherence of final implementation to the initial abstract specification. While refining system model we distinguish between two types of transformations: reduction of non-determinism and extension of system functionality. Following [4] we call these transformations narrowing and supplementing correspondingly.

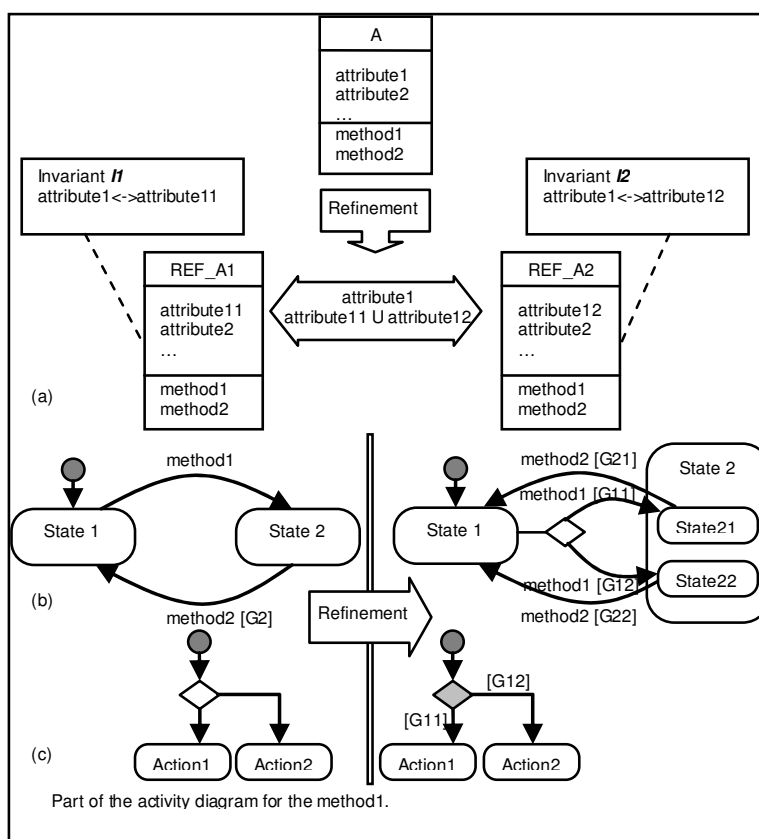


Figure 4. Narrowing pattern

Narrowing focuses on refining generic types of attributes of the system model. Such a transformation results in replacing abstract attributes with the attributes, whose representation is closer to eventual implementation.

The forms of narrowing vary widely and hence are difficult to generalize. However, in the design of protocol the form of narrowing shown in Figure 4 is frequently used. Hence, this transformation can be seen as a generic pattern for narrowing in the field of protocol design.

Essentially the pattern describes the splitting a generic class into several specialized subclasses. The generic class is replaced by the specialized subclasses. Such a narrowing transforms the system model as follows. The classes REF_A1 and REF_A2 refine the class A. They replace A in the class diagram. The invariants I1 and I2 of REF_A1 and REF_A2 respectively, define data refinement relation which describes connection between replaced and newly introduced classes in the class diagram. It is expressed in terms of connecting the attribute representing the identifier of the generic class with the subtypes of that attribute in subclasses. Introduced subclasses have the same methods as a generic class. The associations between generic class and other classes in the class diagram are transformed too. If a relationship should be established on the subtype, the association is split into two new associations connecting one of the subclasses and other class. Otherwise, the invariant of the association is updated so that it connects the union of subtypes instead of the specialized attribute. Observe that only specialized attribute is changed by narrowing while others are staying unchanged (see, e.g., attribute2 in Figure 4(a)). Hence, while transforming state diagram we leave the states and transitions unaffected by narrowing intact, as, e.g., State 1 in Figure 4(b). The states affected by narrowing are transformed into superstates. Such superstates contain states which describe the dynamic behaviour of the system in terms of newly introduced attributes. For instance, State 2 becomes a superstate containing states State 21 and State 22. The transitions, triggered by the invocation of methods affected by narrowing, are also transformed. If they were originally not restricted by the conditions (see, e.g., transition labeled method1) then they become conditional and conditions over newly introduced attributes determine the destination states. If they were conditional originally (see, e.g., transition labeled method2) then conditions are reformulated in terms of newly introduced attributes.

The activity diagrams modeling the methods affected by narrowing are transformed too. The pattern for this transformation is shown in Figure 4(c). Within the transformation nondeterministic branch element is replaced with the sequential or alternative branch. The guards introduced in the state diagrams to define the destination state after the invocation of the corresponding method (see, e.g., method1) are introduced into activity diagram as guards of the deterministic or alternative branch. This reduces the non-determinism of the corresponding method.

Let us demonstrate how requirements change can be handled via narrowing. In the process of protocol development it is necessary to distinguish between the types of packets which node transmits. The modified requirements are as follows:

R2'. Each node can send data and messages it has received

R3'. Each node can receive data and messages

R9'. If a node has a route toward the destination in its routing table, it sends data and messages to the next node along this route. After sending data it waits for other data and messages to be received. When sending messages node buffers them first and then waits for the new information.

Notice that changing one requirement may have the impact on all the existing requirements which include some packet manipulation and hence in the corresponding UML models (listed in the third column of the Table 1). Implemented changes are depicted in the refined diagrams named as: REF number of the refinement step_existing diagram name.

Table 2. System requirements – first refinement

<i>Label</i>	<i>Description of requirement modelling</i>	<i>Used UML diagrams</i>
R2'	Instead of class CPacket we obtain classes CData & CMsg. PacketID attribute is replaced by attributes Data and Msg.	REF1_Class diagram
R3'	The superstate Sending is split into substates SendingData and SendingMsg. The transition from the state Idle to these two new states is also partitioned since the method of this transition – Send – manipulates over attribute PacketID. Thus, Send becomes conditional transition.	REF1_Class diagram REF1_CMobileNode State diagram
R9'	Refined body of the method Send.	REF1_Method Send activity diagram

We omit the demonstration of transformations of class and state diagrams – they can be reconstructed from Table 1.

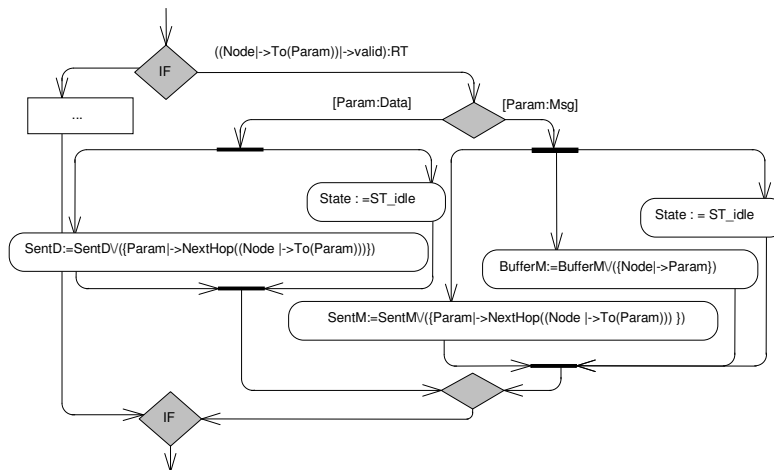


Figure 5. Excerpt from the narrowing of the activity diagram for the method Send

In Figure 5 we illustrate how the activity diagram is transformed by this narrowing according to the pattern described in Figure 4(c).

To guarantee consistency of UML model transformation we should ensure that system specification described by the transformed models is a refinement of a more abstract model. To verify this we again translate the obtained UML model into B and verify refinement between these formal specifications.

The ideas underlying formal stepwise refinement in B and model transformations in MDA coincide: in both cases we aim at advancing implementation while preserving externally observable system behaviour. The results of intermediate refinement steps in B are also machines, called REFINEMENT. Their structure coincides with the structure of abstract machine.

However, refined machine should contain an additional clause REFINES which defines the machine which is refined by the current specification. Besides definitions of variables types the invariant of the refinement machine should contain the refinement relation. This is a predicate which describes the connection between state spaces of more abstract and refined machines.

```

REFINEMENT CMobileNode_R1
REFINES CMobileNode
<specification of machine's sets and variables>
INVARIANT
    Data <: PacketID & Msg <: PacketID &
    Data ∨ Msg = PacketID & Data ∧ Msg = {} &
    SentD <: Sent & SentM <: Sent &
    SentD : Data <-> T_NODE_ID & SentM : Msg <-> T_NODE_ID &
    BufferD <: Buffer & BufferM <: Buffer &
    BufferD : T_NODE_ID <-> Data & BufferM : T_NODE_ID <-> Msg &

<initialisation>
OPERATIONS
Send =
    <specification of the operation body>
    IF
        ( ( Node |-> To ( Param ) ) |-> valid ) : RT
    THEN
        IF
            Param:Data
        THEN
            SentD:=SentDV({Param|->NextHop((Node|->To(Param))}) ||
            State := ST_idle
        ELSIF
            Param : Msg
        THEN
            SentM:=SentMV({Param|->NextHop((Node|->To(Param))}) ||
            BufferM := BufferM ∨ ( { Node |-> Param } ) ||
            State := ST_idle
        ELSE
            skip
        END
    ELSE
        <specification of the operation body – contd>
    END
END

```

Figure 6. Narrowing of the method Send in B

An excerpt from the specification resulting from the translation of our transformed UML models in B is shown in Figure 6. The excerpt illustrates construction of date refinement relation and changes introduced by the refinement into the body of the method Send.

Another typical model transformation is supplementing. While supplementing a model we introduce new features into system functionality while preserving already existing features. We model supplementing by introducing new class with attributes and methods describing the manipulation over new attributes. The connection between classes is an (unidirectional) association which has the same name as the introduced class as shown in Figure 7. In the process of supplementing, previously defined methods can be modified to specify computation over newly introduced attributes and links to new methods.

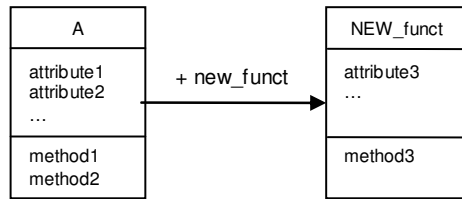


Figure 7. Supplementing pattern

Supplementing presented in Figure 7 demonstrates an introduction of new attribute `attribute3` and computation over it defined by `method3`. This transformation affects the models describing the behaviour of already existing class `A`, i.e., it requires modification of corresponding state and activity diagram. Essentially, such a modification integrates into these models a specification of computation over newly introduced attribute and the link to the new method.

Next we demonstrate how supplementing can facilitate an introduction of a new requirement into the model of AODV protocol. The requirement to be captured is as follows:

R10: To ensure that the routing information is fresh enough and to guarantee loop free routes, each node maintains a sequence number (SN). The sequence number is incremented every time when the node sends a message to initiate discovery of a missing route.

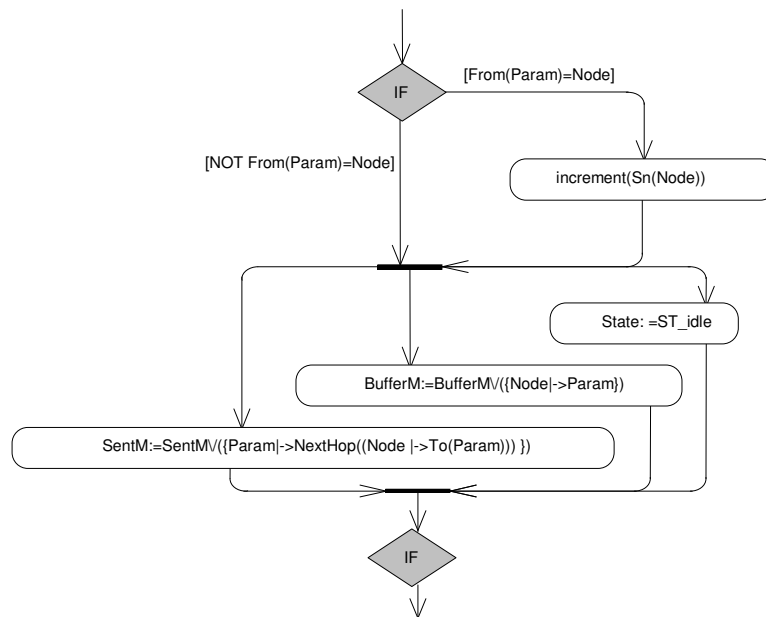


Figure 8. Supplementing in the activity diagram

An introduction of this requirement via supplementing, results in creating a new class `CSn` with the attribute `Sn` and the method `increment`.

We transform the activity diagram of the method `Send` by inserting activity elements describing computation over `Sn` as shown in Figure 8. The excerpt from the corresponding B specification capturing this modification is shown in Figure 9.

```

REFINEMENT CMobileNode_R2
REFINES CMobileNode_R1
  <specification of machine's sets and variables>
  INVARIANT
    Sn : { Node } --> NAT &
  INITIALISATION
    Sn := { ( IP_address |-> 0 ) } ||
  OPERATIONS
  Send=
    <specification of the operation body>
    ELSIF
      Param : Msg
    THEN
      IF
        From ( Param ) = Node
      THEN
        increment (Sn(Node)) ;
        SentM := SentM / ( { Param } -> NextHop ( Node |-> To ( Param ) ) ) ||
        BufferM := BufferM ∨ ( { Node |-> Param } ) ||
        State := ST_idle
      ELSE
        SentM := SentM / ( { Param } -> NextHop ( Node |-> To ( Param ) ) ) ||
        BufferM := BufferM ∨ ( { Node |-> Param } ) ||
        State := ST_idle
      END
    ELSE skip
    END
    <specification of the operation body-contd>
  END

```

Figure 9. Supplementing in the method `Send`

The further refinement steps result in differentiating between types of messages and replacing sets with more concrete data structures. Namely the messages are split into RREQ and RREP subsets (B specification of this step for the method `Send` can be found in Appendix - Figure A4). Replacement of sets with concrete data structures allows us to specify the way in which exchanging packets are to be handled, i.e., packets are sent in the FIFO manner – first packet received, first sent.

While presenting the case study, we focused on examples illustrating the requirements evolution and change. In this section we demonstrated how the new and changing requirements are introduced into a system model in a systematic and correctness preserving way.

5. Conclusion

This paper has presented an approach to handling requirements changes and evolution in formalized model-driven development. We proposed patterns for refining UML models to implement changing or new requirements. Moreover, we described formal semantics in B for refinement of class, state and activity UML diagrams. We demonstrated how UML modeling combined with formal specification can improve requirements traceability and support navigation through the design space. The proposed approach was validated by a case study – development of AODV routing protocol for mobile ad-hoc networks.

Correctness preserving development in UML has been studied also by Liu et al. [5]. They showed how the stepwise refinement of UML models supports the maintenance of consistency during model transformation and evolution. Their UML model of a system consists of a class, use case, sequence and state diagrams. They do not consider activity diagrams since the computation which methods specify is assumed to be simple. In our work, we demonstrated that capturing system requirements with activity diagrams is especially useful when complex computation should be modeled.

Varro and Pataricza have also proposed an approach supporting model transformations in the MDA environment [14]. They presented a visual yet formal specification technique based on metamodeling and graph transformations. They demonstrated how to automatically implement model transformations specified on a very abstract level based on transformation rules and mappings of UML on Action Semantics. Our approach uses the B Method as a formal framework and allows to address modeling at different levels of abstraction.

The development conducted in this paper was supported by Atelier B – an automatic tool for verification and refinement in B. The tool support has significantly simplified the development process and increased our confidence in the correctness of obtained models. We believe that the availability of the tool supporting formal specification and verification as well as tight integration with UML can facilitate acceptance of our approach in industry.

As a future work it would be interesting to explore the use of OCL instead of B notation to define logical conditions in UML modeling. Moreover, it would be useful to integrate the formalized model-driven development presented in this paper with simulators used to estimate performance of ad-hoc protocols. In this case we could address not only correctness but also performance issues in the development process.

References

- [1] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
- [2] J. R. Abrial. Event Driven Sequential Program Construction, 2001. <http://www.atelierb.societe.com/ressources/articles/seq.pdf>
- [3] ClearSy, Aix-en-Provence, France. *Atelier B - User Manual*, Version 3.6, 2003.
- [4] R. Kobro Runde, “Refining UML interactions”, In *Proceedings of the 16th Nordic Workshop on Programming Theory*, Uppsala University, Sweden, Oct 2004, pp: 36-38.
- [5] Z. Liu, X. Li, J. Liu and H. Jifeng, “Integrating and Refining UML Models”, UNU-IIST Technical Report No. 295, March 2004. <http://www.iist.unu.edu/newrh/III/1/docs/techreports/report293.pdf>
- [6] Z. Liu, X. Li, J. Liu and H. Jifeng, “Linking UML Models of Design and Requirement”, Australian Software Engineering Conference (ASWEC'04), Melbourne, Australia, April 2004, pp: 329-339.
- [7] *MATISSE Handbook for Correct Systems Construction*. EU-project MATISSE: Methodologie and Technologies for Industrial Strength Systems Engineering, IST-199-11345, 2003. <http://www.esil.univ-mrs.fr/~spc/matisse/Handbook>
- [8] J. Miller and J. Mukerji (Editors), *MDA Guide Version 1.0.1*, 2003. <http://www.omg.org/docs/omg/03-06-01.pdf>
- [9] C. E. Perkins, E. M. Belding-Royer and I. Chakeres, “Ad Hoc on Demand Distance Vector (AODV) Routing”, *IETF Internet draft*, 2003. <http://moment.cs.ucsb.edu/pub/draft-perkins-manet-aodvbis-00.txt>
- [10] J. Rumbaugh, I. Jacobson and G. Booch, *Unified Modeling Language Reference Manual*, Addison Wesley, 1999.
- [11] S. Schneider, *The B Method. An introduction*, Palgrave, 2001.
- [12] Snook, C. Combining UML and B. In *Proceedings of Forum on specification & design languages*, Marseille, 2002.
- [13] *U2B Manual for U2B Version 3.6.8*, University of Southampton <http://www.ecs.soton.ac.uk/~cfs/U2Bdownloads/U2Bevaluation/U2BManual.pdf>
- [14] D. Varro and A. Pataricza, “UML Action Semantics for Model Transformation Systems”, *International Journal of Periodica litechnica*, 2003

Appendix

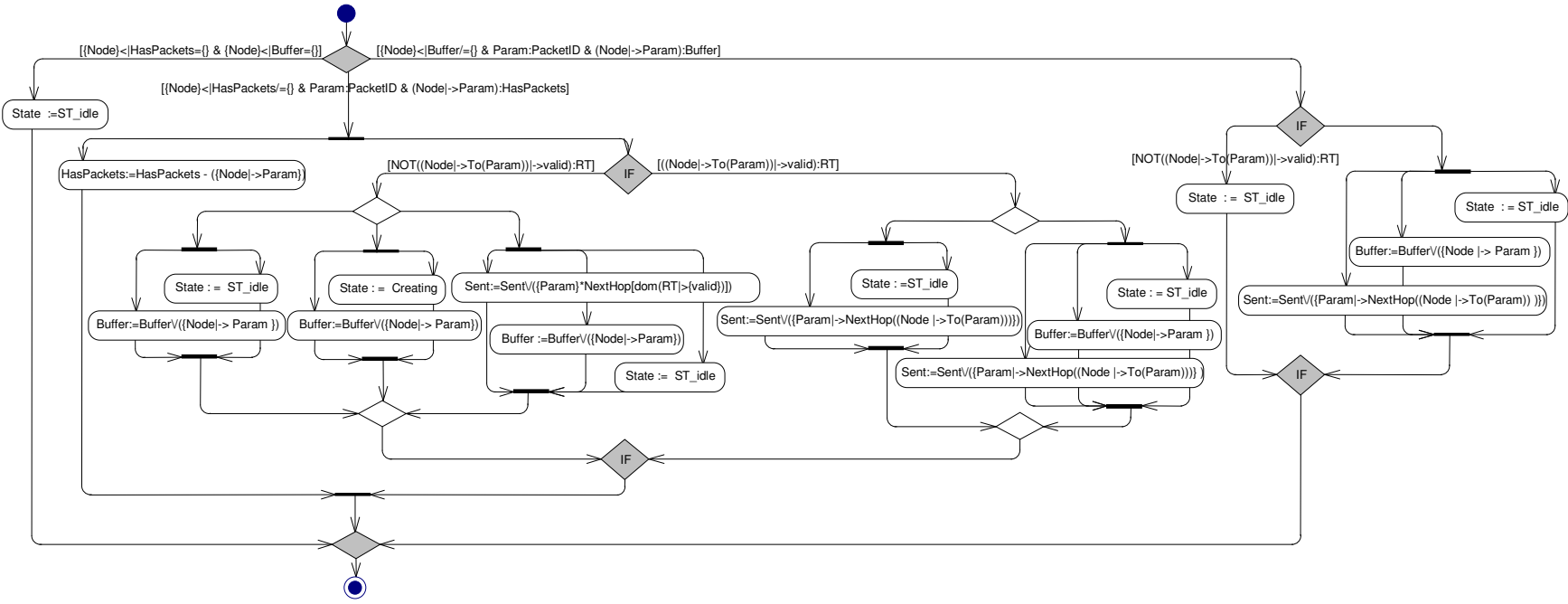


Figure A1. Complete activity diagram of the method Send


```

MACHINE                               Global
SETS                                  STATES = { ST_Idle , Sending , Receiving , Creating } ;
                                       STATUS = { valid , invalid , unknown }
ABSTRACT_CONSTANTS
                                       T_NODE_ID ,
                                       T_ROUTE ,
                                       T_PACKET_ID ,
                                       IP_address
PROPERTIES
                                       T_NODE_ID <: NAT1 &
                                       T_ROUTE : T_NODE_ID <-> T_NODE_ID &
                                       T_PACKET_ID <: NAT1 &
                                       IP_address : T_NODE_ID
END

```

Figure A2. a) Machine defining types

```

MACHINE                               CRoute
SEES                                   Global
VARIABLES
                                       Route,
                                       RT,
                                       NextHop
INVARIANT
                                       Route<:T_ROUTE &
                                       RT : Route --> STATUS &
                                       NextHop : Route --> T_NODE_ID
INITIALISATION
                                       Route:={} || RT:={} || NextHop:={}
END

```

Figure A2. b) CRoute machine

```

MACHINE                               CPacket
SEES                                   Global
VARIABLES
                                       PacketID,
                                       From,
                                       To
INVARIANT
                                       PacketID <: T_PACKET_ID &
                                       From : PacketID --> T_NODE_ID &
                                       To : PacketID --> T_NODE_ID
INITIALISATION
                                       PacketID:={} || From:={} || To:={}
END

```

Figure A2. c) CPacket machine

```

Send=
  SELECT    CMobileNode_State=Sending & {Node}<|HasPackets ={} &
           {Node}<|Buffer={}
           /*If the node doesn't have any packets
           to send it becomes idle.*/
  THEN
    CMobileNode_State:=ST_Idle
  WHEN
    CMobileNode_State=Sending & {Node}<|HasPackets /={} &
    Param:PacketID & (Node|->Param):HasPackets
  THEN
    HasPackets:=HasPackets - ((Node|->Param)) ||
    IF
      ((Node|->To(Param))|->valid):RT
      /* Checks if the destination exists in the
      routing table of the sending node.*/
    THEN
      Sent:=Sent\/{Param|->NextHop((Node|->To(Param)))} ||
      CHOICE
        CMobileNode_State:=ST_Idle
      OR
        Buffer:=Buffer\/{Node|->Param)} ||
        CMobileNode_State:=ST_Idle
      END
    ELSE
      Buffer:=Buffer\/{Node|->Param)} ||
      CHOICE
        CMobileNode_State:=Creating
      OR
        CMobileNode_State:=ST_Idle
      OR
        Sent:=Sent\/{Param}*NextHop[dom(RT|>{valid})]} ||
        CMobileNode_State:=ST_Idle
      END
    END
  WHEN
    CMobileNode_State=Sending & {Node}<|Buffer/={} &
    Param:PacketID & (Node|->Param):Buffer
  THEN
    IF
      ((Node|->To(Param))|->valid):RT
    THEN
      Buffer:=Buffer-({Node|->Param)} ||
      Sent:=Sent\/{Param|->NextHop((Node|->To(Param)))} ||
      CMobileNode_State:=ST_Idle
    ELSE
      CMobileNode_State:=ST_Idle
    END
  END;

```

Figure A3. B specification of the method Send (obtained by translating the diagram from the Figure A1)

```

Send =
SELECT
    CMobileNode_State=Sending & {Node}<|HasPackets={} &
    {Node}<|BufferD={} & {Node}<|BufferM= {}
THEN
    CMobileNode_State:=ST_Idle
WHEN
    CMobileNode_State=Sending & {Node}<|HasPackets/={} &
    Param:Data\RREQ\RREP & (Node|->Param):HasPackets
THEN
    HasPackets:=HasPackets-({Node|->Param}) ||
    IF
        ((Node|->To(Param))|->valid):RT
    THEN
        IF
            Param:Data
        THEN
            SentD:=SentD\({Param|->NextHop((Node|->To(Param)))}) ||
            CMobileNode_State:=ST_Idle
        ELSIF
            Param:RREQ
        THEN
            SentM:=SentM\({Param|->NextHop((Node|->To(Param)))}) ||
            IF
                From(Param)=Node
            THEN
                Sn(Node):=Sn(Node)+1;
                SnM(Param):=Sn(Node) ||
                BufferM:=BufferM\({Node|->Param}) ||
                State:=ST_Idle
            ELSE
                BufferM:=BufferM\({Node|->Param}) ||
                CMobileNode_State:=ST_Idle
            END
        ELSIF
            Param:RREP
        THEN
            SentM:=SentM\({Param|->NextHop((Node|->To(Param)))}) ||
            CMobileNode_State:=ST_Idle
        ELSE
            Skip
        END
    ELSE
        IF
            Param:Data
        THEN
            BufferD:=BufferD\({Node|->Param}) ||
            CMobileNode_State:=Creating
        ELSIF
            Param:RREQ
        THEN
            BufferM:=BufferM\({Node|->Param}) ||
            SentM:=SentM\({Param}*NextHop[dom(RT)|>{valid}]) ||
            CMobileNode_State:=ST_Idle
        ELSIF
            Param:RREP
        THEN
            BufferM:=BufferM\({Node|->Param}) ||
            CMobileNode_State:=ST_Idle
        ELSE
            Skip
        END
    END
WHEN
    CMobileNode_State=Sending & {Node}<|BufferD/={} &
    Param:Data\RREQ\RREP & (Node|->Param):BufferD\BufferM
THEN

```

```

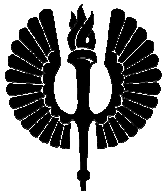
IF
  ((Node|->To(Param))|->valid):RT
THEN
  IF
    Param:Data & (Node|->Param):BufferD
  THEN
    BufferD:=BufferD-({Node|->Param}) ||
    SentD:=SentD\({Param|->NextHop((Node|->To(Param)))}) ||
    CMobileNode_State:=ST_Idle
  ELSIF
    Param:RREQ\RREP & (Node|->Param):BufferM
  THEN
    BufferM:=BufferM-({Node|->Param}) ||
    SentM:=SentM\({Param|->NextHop((Node|->To(Param)))}) ||
    CMobileNode_State:=ST_Idle
  ELSE
    Skip
  END
ELSE
  CMobileNode_State:=ST_Idle
END
END;

```

Figure A4. Refinement of the method Send – introduction of two kinds of messages: RREQ and RREP

TURKU
CENTRE *for*
COMPUTER
SCIENCE

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research



Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 952-12-1507-0
ISSN 1239-1891