STF90 A04042 – Unrestricted

REPORT

Graphical Specification of Dynamic Network Structure

Fredrik Seehusen and Ketil Stølen

SINTEF ICT

June 2004

SINTEF

SINTEF ICT

Address: NO-7465 Trondheim
NORWAY
Location Trondheim:
S.P. Andersens v 15
Location Oslo:
Forskningsveien 1
Telephone: +47 73 59 30 00
Fax: +47 73 59 43 02

Enterprise No.: NO 948 007 029 MVA

SINTEF REPORT

TITLE

Graphical Specification of Dynamic Network Structure

AUTHOR(S)

Fredrik Seehusen and Ketil Stølen

CLIENT(S)

REPORT NO.	CLASSIFICATION	CLIENTS REF.	
STF90 A04042	Unrestricted		
CLASS. THIS PAGE	ISBN	PROJECT NO.	NO. OF PAGES/APPENDICES
Unrestricted	82-14-03107-9	40332800	18
ELECTRONIC FILE CODE		PROJECT MANAGER (NAME, SIGN.)	CHECKED BY (NAME, SIGN.)
rapport_forside		Ketil Stølen	Mass Soldal Lund
FILE CODE	DATE	APPROVED BY (NAME, POSITION, SIGN.)	
	2004-06-03	Bjørn Skjellaug, Research director	

ABSTRACT

We present a language, MEADOW, for specifying dynamic networks from a structural viewpoint. We demonstrate MEADOW in three examples addressing dynamic reconfiguration in the setting of object-oriented networks, ad hoc networks and mobile code networks. MEADOW is more expressive than any language of this kind (e.g. SDL-2000 agent diagrams, composite structures in UML 2.0) that we are aware of, but maintains, in our opinion, the simplicity and elegance of these languages.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	ICT, modelling	IKT, modellering
GROUP 2	Design, component, mobility, network	Design, komponent, mobilitet, nettverk
SELECTED BY AUTHOR	UML, SDL, language	UML, SDL, språk
	Dynamic network	Dynamisk nettverk

Graphical Specification of Dynamic Network Structure

Fredrik Seehusen and Ketil Stølen
Department of Informatics, University of Oslo, Norway
SINTEF ICT, Norway
{*fredrik.seehusen,ketil.stoelen*}*@sintef.no*

Abstract

We present a language, MEADOW, for specifying dynamic networks from a structural viewpoint. We demonstrate MEADOW in three examples addressing dynamic reconfiguration in the setting of object-oriented networks, ad hoc networks and mobile code networks. MEADOW is more expressive than any language of this kind (e.g. SDL-2000 agent diagrams, composite structures in UML 2.0) that we are aware of, but maintains, in our opinion, the simplicity and elegance of these languages.

1 Introduction

Many state-of-the-art graphical languages such composite structures in UML, are well suited to describe the static structure of networks. Today however, networks which exhibit dynamic reconfiguration such as object-oriented networks [8], ad hoc networks [1, 2, 9] and mobile code networks [5] are of great practical importance, and many languages lack concepts with which to express the dynamic aspects of these networks. Traditionally, behavior and dynamic characteristics are not expressed in structure diagrams, i.e. behavior is expressed with notations such as state machines or sequence diagrams.

In this report, we aim to address this by proposing concepts with which to describe dynamic reconfiguration in structural diagrams through the introduction of new graphical language, MEADOW (ModElling lAnguage for DataflOW). MEADOW distinguishes between diagrams for specifying *snapshots* of the network structure at one point in time, and diagrams for specifying the *potential* structure that networks may exhibit over a period of time. A diagram of potential structure contains information about dynamic reconfiguration in the sense that it constraints the potential structure that a network may exhibit during its lifetime. Snapshot diagrams may be used to model the structure that a network may exhibit at the time of instantiation, and at later points in time. Snapshot diagrams can be related in time to model dynamic reconfiguration. In addition, MEADOW distinguishes clearly between static and dynamic constructs in order to capture both static and dynamic aspects of networks.

The contributions of this report are:

- The definition of central network concepts and dynamic networks.

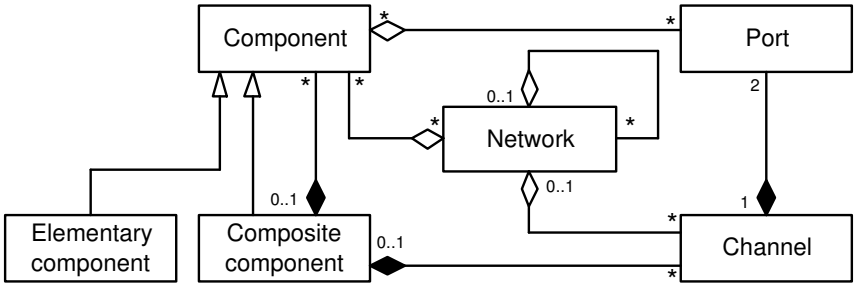

Figure 1: Basic network concepts

- The presentation of concepts with which to express dynamic reconfiguration through an example-driven introduction of the language MEADOW.

The rest of the report is structured as follows. In section 2, we define the basic network related concepts used in this report. Section 3 aims to provide sufficient background on MEADOW to allow the examples to be understood. In section 4, 5 and 6 we employ MEADOW to specify the structure of an object-oriented network, an ad hoc network and a mobile code network, respectively. Section 7 describes related work and section 8 concludes this report.

2 Network Concepts

In order to define a small set of concepts with which to describe networks, we abstract from the distinction of physical and logical layers. We say that a network is a set of *components* and a set of *channels* over which the components communicate. Networks that consist of computers that are connected by fixed wires, or networks that consist of application processes connected by TCP/IP are thus represented in the same way.

In the following we (1) define basic network concepts (2) use these concepts to define three kinds of dynamic networks: object-oriented networks, ad hoc networks and mobile code networks.

2.1 Networks

The UML class diagram in figure 1 describes the basic network concepts that are used in this report. As expressed in the diagram, a network may consist of components, channels and (sub) networks. We have distinguish between two types of components: *elementary components* and *composite components.* Elementary components do not contain components (referred to as sub-components), while the composite components contain sub-components and channels over which the sub-components communicate. Each *channel* has exactly two *ports.* Components reference these ports in order to receive or transmit messages on channels. These references represent interfaces between a component and its environment. The ports are associated with channels and not components (as they are in UML for instance). This makes it easier to use formalisms that are based on π-calculus [10] to specify the behavior of components.

The terms defined below are based on well-established constructs from existing languages and methods [3, 6, 7, 11].

Network A set of components, a set of channels over which the components communicate, and a set of sub-networks. We distinguish between a *network type* and a *network instance*. A network type defines a set of common features shared by all of its instances, whereas a network instances has its own identity and its own set of properties that conforms to the features defined by its type.

Component An entity that communicates with its environment through a set of referenced ports. A component may be sent from one component to another via channels. A component has a *behavior* which defines (1) how messages that are received by its referenced ports are handled and (2) how messages are output on its referenced ports. We distinguish between a *component type* and a *component instance*. A component type defines a set of common features shared by all of its instances. Each component instance (1) is of a specific type, (2) has its own identity and its own set of properties that conforms to the features defined by its type. A component can be elementary or composite.

Elementary Component A component that can not contain sub-components or channels.

Composite Component A component that may contain (reference) sub-components and channels over which the sub-components may communicate. A composite component may communicate with its sub-components.

Port A port provides an interface between a component and its environment. A port is either an *input-port* or an *output-port*. The former receives messages from a channel, whereas the latter transmits messages along a channel. A reference to a port may be sent from one component to another via a channel.

Channel A channel represents the forwarding of messages from an output-port to an input-port, hence a channel is *directed*. A channel is *shared* if any of the ports it connects are referenced by more than one component. A channel may or may not allow message overtaking, message disappearance, and message duplication.

2.2 Static, Dynamic and Mobile Networks

Based on the terms introduced in the previous subsection, we define what we mean by static and dynamic networks. The following definitions rely on the previous terms, hence the following terms are defined with respect to a *model* of a network. Consequently, whether we say that a network is dynamic or not depends on how the network is specified using the previously defined terms, and not necessarily on the network itself (das Ding an sich).

The terms defined below are understood in various ways depending on context and research field. Our definitions are not meant to cover or encompass all of these understandings, but rather to make precise what we mean by these concepts in MEADOW and in this report.

Static network A network is *static* if the sets of references to ports and sub-components of all its components remain constant throughout any computation. Hence, in a static network, components and channels are neither created nor killed during computation.

Dynamic network A network is *dynamic* if (1) the sets of references to ports and subcomponents of one or more of its components does not remain constant throughout a computation or (2) the set of components that are part of the network does not remain constant throughout a computation.

Object-oriented network A dynamic network in which (1) each component references a single input-port and may reference many output-ports, and (2) references to output-ports may be sent along the channels. In other words, a component represents an object, and the single input-port referenced by the component represents a unique object identifier. The output-ports referenced by a component represent object identifiers of other components (pointers) that the component is aware of. The fact that references to output-ports may be sent along the channels represents object identifiers (pointers) that may be passed on from one object to another.

Ad hoc network A dynamic network in which (1) each of the components in its set may be removed from that set during computation, i.e. every component may enter and leave the network during the lifetime of the network and (2) all the components may change their communication partners during the lifetime of the network. An object-oriented network is a special case of an ad hoc network.

Mobile code network A network which enables a component (representing the mobile code) to be sent on a channel from one (composite) component to another (composite) component.

3 MEADOW Basics

In MEADOW, there is no graphical notation for networks, components and channels. Instead they are modeled by so-called *regions*, *parts* and *connectors*[1], respectively.

Assume that I is a set of identifiers, n is a natural number, and that T and i are identifiers. Unless otherwise specified, the following applies throughout the rest of the report:

- :T denotes either (1) a component type named T or (2) a network type named T.
- T denotes an unnamed instance of :T.
- T[n] denotes n unnamed instances of :T.
- i:T denotes an instance named i of :T.
- I:T denotes e:T for each e $\in$ I.

We briefly describe the constructs of parts and connectors. The graphical notations of regions, parts, connectors and other constructs are indicated in figure 2. For more details, please consult [14].

A part is a subset of the set of all instances of a component type. The graphical notation for a part is a box.

[1]These constructs are similar to parts and connectors in UML 2.0, and we saw no reason to name them differently.

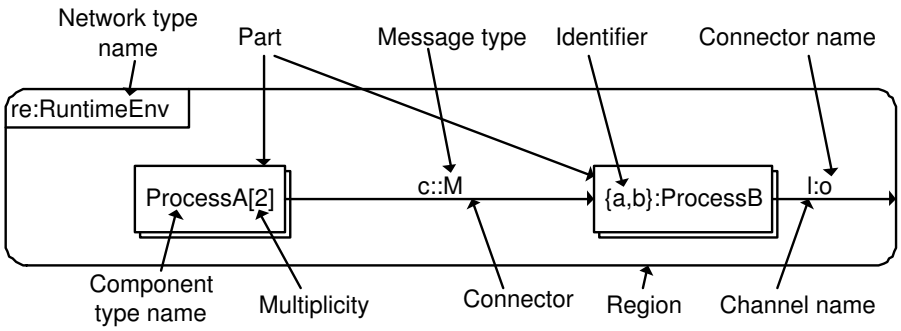

Figure 2: RuntimeEnv

A region is a subset of the set of all instances of a network type. The graphical notation for a region is a box with rounded edges.

A connector is a set of channels. Connectors connect parts as channels connect component instances. The graphical notation for a connector is a directed or bidirected arrow.

Regions, parts and connectors have a minimum and a maximum cardinality that may be defined by identifiers and/or multiplicities. In figure 2, the part labeled ProcessA[2] has a minimum cardinality of two and a maximum cardinality of two as defined by the multiplicity ([2]). Note that connectors are associated with a label of the form ch:co::m, where ch denotes a channel name, co denotes a connector name, and m denotes a message type.

Regions, parts and connectors are either *static* or *dynamic*. Graphically, static constructs have a solid outline, while dynamic constructs have a dashed outline. E. g. the notation of a static part is a box with a solid outline, and the notation of a dynamic part is a box with dashed outline.

The two most important kinds of diagrams in MEADOW are the type diagram and the snapshot diagram. Type and snapshot diagrams define the features of either component types or network types in terms of parts, regions and connectors. We explain each of these diagrams in turn.

A type diagram is a specification of the potential structure that all instances of either a component type or a network type may exhibit during their lifetime. Let S be either a static region, a static part or a static connector, and D be either a dynamic region, a dynamic part or a dynamic connector. A type diagram that (1) defines the features of :T and (2) contains S and D, specifies that an instance i of :T must contain (1) exactly the same instances of S at all times during the lifetime of i:T and (2) a varying number of the instances of D during the lifetime of i:T.

A snapshot diagram is a specification of the structure that all instances of a either component type or a network type may exhibit at one or more points in time. Let S be either a static region, a static part or a static connector. A snapshot diagram that (1) defines the features of :T and (2) contains S, specifies that an instance i of :T must contain the specified number of the instances in S at zero or more points in time. A snapshot diagram can not contain dynamic constructs.

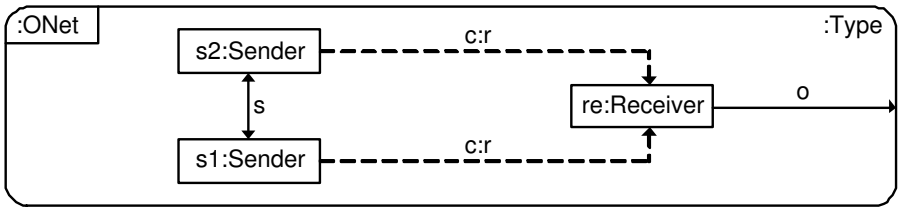

Figure 3: Type diagram of ONet

Sometimes, a model of a component/network type consists of more than one diagram. Thus it is desirable to structure all the diagrams that logically belong together. This brings us to the third kind of diagram that exists in MEADOW, the views-diagram.

A views diagram structures all the diagrams that a model of a component/network type consists of. It is a set of diagram declarations and it may classify these into views. A views diagram may also contain constructs of relating declarations of snapshot diagrams in time. We shall see examples of this in the next sections.

4 Object-oriented Network: ONet

We model an object-oriented network called ONet. ONet consists of three objects: two objects of class Sender and one object of class Receiver. Each sender object always has a reference to the other sender object, and one and only one sender object has a reference to the receiver at any given time. The sender objects may exchange the reference to the receiver with each other in order to send messages to it.

We model ONet as the network type :ONet. The model of the network consists of four diagrams. We shall explain each of these in turn.

4.1 ONet:Type

The first diagram is presented in figure 3. As defined in the top right corner of the diagram, figure 3 presents a *type diagram*. As explained previously, a type diagram contains the specification of the potential structure that all instances of a component type or a network type may exhibit during their lifetime. In the current example, this means that every instance of the network type :ONet, must have the potential structure specified in figure 3.

The parts labeled s1:Sender and s2:Sender consist of one component instance named s1 of :Sender and one instance named s2 of Sender, respectively. Similarly, the part labeled re:Receiver consists of one component instance named re of :Receiver. The three parts in figure 3 are all *static*, because the graphical notation of a static part is a box with a solid outline. A static part must always consist of the same instances during its lifetime. In the current example, this means that each instance n of :ONet, must contain s1:Sender, s2:Sender, and re:Receiver at all times during the lifetime of n. Note that no two network instances of :ONet contain the *same* instances of :Sender and :Receiver even though they contain instances with the same names and the same component types.

The diagram in figure 3 contains two *static connectors* named s and o and two *dynamic connectors* labeled c:r. The graphical notation of static connector is an arrow

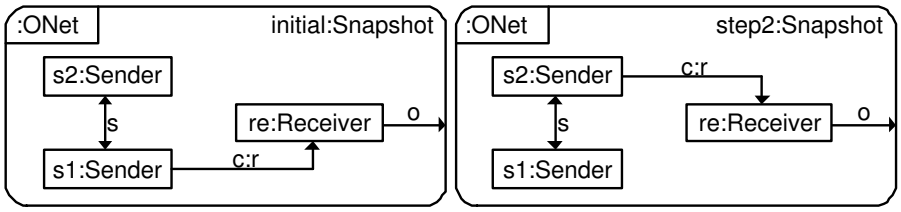

Figure 4: Snapshot diagrams of ONet

with a solid outline, whereas the notation of a dynamic connector is a dashed arrow. A static connector consists of channels that are always part of a composite component or a network as long as the components the channels connect are part of the same composite component or network. A dynamic connector, however, is a connector that consists of channels that may or be created or killed *during* the lifetime of the component instances they connect.

The relationship of channels that a connector consists of is affected by the cardinality of the parts that the connector connects. In general, a connector c going from a part P_1 to a part P_2 means that $\forall x \in P_1, \forall y \in P_2$, there is a channel from component instance x to component instance y.

A connector may be directed or bi-directed. The graphical notation of a directed connector is a line with an arrowhead at one end, whereas the graphical notation of a bi-directed connector is a line with an arrowhead in both ends. A bi-directed connector is two directed connectors containing channels that forward messages in opposite directions. In the current example, the connector named s is bi-directed. It consists of two channels: one channel going from s1:Sender to s2:Sender, and one channel going in the opposite direction. Since the connector s is static, each network n of the network type :ONet must contain two channels that connect the two instances of :Sender at all times during the lifetime of n.

The directed connector named o consists of one channel that goes from re:Receiver to the environment of the network that re:Receiver is a part of. In other words, each network n of :ONet must have a channel that goes from re:Receiver to the environment of n at all times during the lifetime of n.

The lowermost dynamic connecter labeled c:r is named r, and it consists of one channel named c that goes from s1:Sender to re:Receiver. The fact that the connector is dynamic means that the channel the connector consists of may or may not be part of a network n of :ONet during the lifetime of n even if both components (s1 and re) are part of n. The fact that both dynamic connectors are equally named and consists of a channel with the same name (c), implies that the channel c can not be part of both connectors at a given point in time. That is, the specification given in figure 3 is such that only one component instance may transmit messages on channel c, and only one component instance may input message on c at any given time.

4.2 ONet:Snapshots

A *snapshot diagram* specifies the configuration that instances of a component type or a network type may exhibit at *a point in time*.

Our model of ONet consists of the two snapshot diagrams presented in figure 4. The leftmost diagram is named initial and the rightmost diagram is named step2. The name

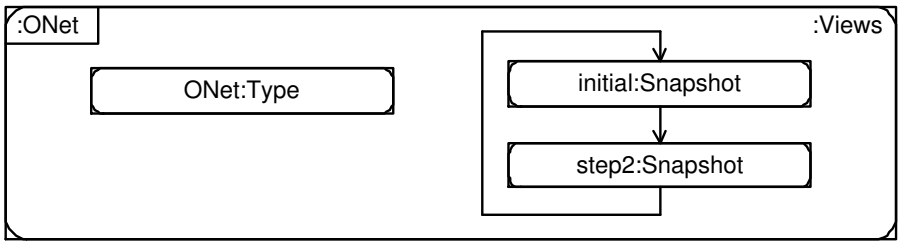

Figure 5: Views diagram of ONet

initial is reserved. A snapshot diagram named initial specifies the initial structure that all instances of a component type or a network type must exhibit upon their creation. In this example, all networks of type :ONet must exhibit the structure specified in the leftmost diagram in figure 4 upon their creation.

The diagram named step2 specifies the structure that all networks of :ONet may exhibit at a point in time after they have been created.

4.3 ONet:Views

The last of the four diagrams that the model of ONet consists of is presented in figure 5. As indicated in the top right corner of the diagram, this is a so-called *views diagram*. The diagram in figure 5 contains the declaration of the three diagrams we have seen so far, that is ONet:Type (figure 3), inital:Snapshot (figure 4) and step2:Snapshot (figure 4).

The arrows in the diagram specify how the snapshot diagrams are related in time. Let $D_1 \rightarrow D_2$ denote an arrow from diagram D_1 to diagram D_2 for a network type :N. $D_1 \rightarrow D_2$ means that a network n of :N may exhibit the structure specified in D_2 after it has exhibited the structure specified in D_1. If $D_1 \rightarrow D_2 \rightarrow D_3$, then n may first exhibit D_1, then D_2, then D_3. However, n may not go from exhibiting the structure specified in D_1 to D_3 without first exhibiting the structure specified in D_2. Thus the constructs for specifying how snapshot diagrams are related in time are not only a way of increasing the understandability appropriateness of a model, they also provide a basis for model checking. This can be achieved by comparing the specification of how snapshot diagrams are related in time with a specification of the behavior of components.

The type diagrams may also be used as a basis for a similar kind of model checking. One can check if a network exhibits a structure during computation that is not specified in the type diagram. Such an occurrence may indicate flaws in the specification of component behavior, provided that the type diagram is correct.

5 Ad hoc Network: Battlefield Control System

In the following we model a network called the Battlefield Control System (BCS) on the basis of an informal description given in [4]. The system is used to control the movement, strategy and operations of troops in the battlefield.

BCS consists of a commander and a number of soldiers. The commander acts as a server and the soldiers act as its clients. One of the soldiers acts as backup. Upon failure of the commander, the backup will take over as the new commander. Communication

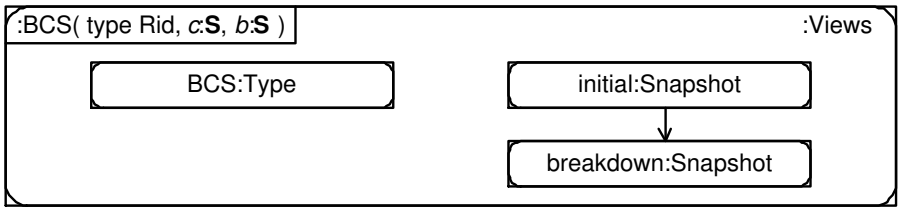

Figure 6: Views diagram of BCS

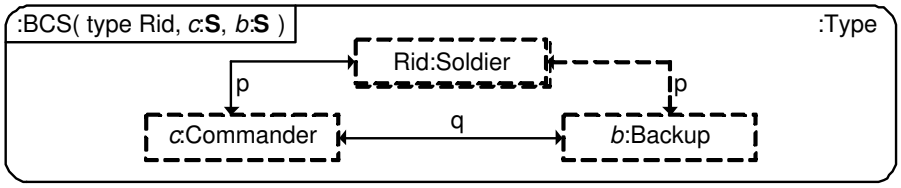

Figure 7: Type diagram of BCS

between clients and the server is only through encrypted messages sent via a radio modem. The radio modem uses a shared communication channel; only one component can broadcast at any moment.

BCS is an ad hoc network in the sense that (1) all components and channels in the network may be created or killed during computation and (2) it is dynamically reconfigurable.

We do not to give a full specification of BCS, but model the scenario of commander breakdown.

5.1 The Views Diagram

We model the system at a logical level (we do not consider the physical level) as the network type :BCS. The three diagrams that specify the internal structure of the network instances of :BCS are declared in the views diagram presented in figure 6.

Note that three formal parameters are declared in the top left corner of the diagram. The first is a type named Rid, the second and third are the constants c and b of type $\mathbb{S}$ (the set of all strings).

5.2 BCS:Type

The type diagram for :BCS is presented in figure 7. All parts in the diagram are *dynamic*, because the graphical notation of this part is a box with dashed outline. In the current example, the commander is modeled as c:Commander. The part that is associated with this label, has a maximum cardinality of one and a minimum cardinality of zero. Here, this means that a network n of network type :BCS may contain zero or one instances of :Commander at any given point during its lifetime. A similar constraint applies for the part that models the backup.

As declared in the top right corner of the diagram in figure 7, Rid is a type, i.e. a set of identifiers. This type is used as an identifier for the part labeled Rid:Soldier, which models the soldiers. The maximum cardinality of this part is #Rid (the cardinality of Rid), and the minimum cardinality is zero. The maximum and minimum cardinality

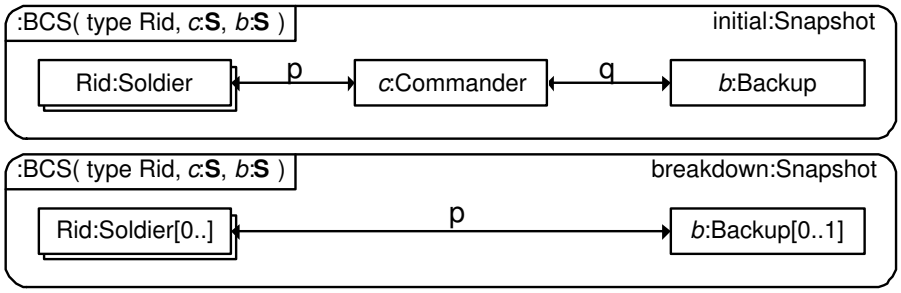

Figure 8: Snapshot diagrams of BCS

of a part could also be determined by a multiplicity. E.g. if, $l = 3$ and $u = 8$, then a part labeled Rid:Soldier[l..u] would have a minimum cardinality of 3 and a maximum cardinality of 8.

The connector named p that connects c:Commander to Rid:Soldier constitutes (1) a one to many relationship of channels from the commander instance to the soldier instances and (2) a many to one relationship of channels from the soldiers to the commander. Note that (1) if both the commander and the soldiers are part of a network n of :BCS, they *must* be connected and (2) if both the backup and the soldiers are part of n, they *may* be connected.

5.3 BCS:Snapshots

Figure 8 presents the two snapshot diagrams that are declared in the views diagram (figure 6). The diagram named initial specifies the initial structure that a network of :BCS must exhibit upon its creation. Notice here that there is no initial connection between the backup and the soldiers.

Diagram breakdown specifies the structure of the network instances of :BCS upon breakdown of the commander. Here, a connection is established between the soldiers and the backup as a result of the commander not being part of the network anymore. The multiplicity ([0..]) associated with the part of type Soldier, specifies that the part has a minimum cardinality of zero and a maximum cardinality of #Rid (#Rid because the upper bound of the multiplicity is not defined) at the time of breakdown. Similarly, the part of type Backup has a minimum cardinality of zero and a maximum cardinality of one at the time of breakdown.

6 Mobile Code Network: PDANet

In this example we model a network that is based on a framework for building context-aware applications in ubiquitous and mobile computing settings. The goal of the framework is to offer a location-aware system in which spatial regions can be determined to within a few square feet, so that one or more portions of a room or a building can be distinguished.

The framework consists of two parts: (1) mobile agents and (2) local information servers, called LISs. The former offers application-specific services which are associated with physical entities and places. The latter provide a layer between underlying

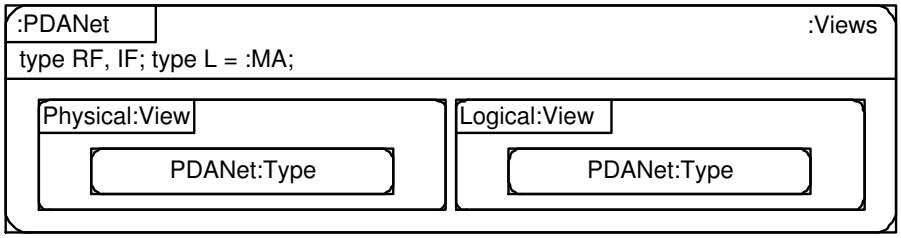

Figure 9: Views diagram of PDANet

locating systems and mobile agents. Each LIS provides the agents with up-to-date information on the state of the real world, such as the locations of people, places and things, and the destinations that agents should migrate to. For a more detailed description of the framework, we refer to [13].

We model a simple network called PDANet that is based on this framework. The physical components that constitutes PDANet are, a sensor, a LIS, a computer associated with a tag and a personal digital assistant (PDA) also associated with a tag. The tags make it possible for the sensor to locate the physical entities (the computer and the PDA). Each tag periodically transmits a unique identifier via infrared light that can be received by the sensor. The tag associated with the computer is always within the presence of the sensor, while the tag associated with the PDA may or may not be within the presence of the sensor at a given point in time. The sensor uses a radio frequency to notify the LIS of the tags that are within the presence of the sensor at a given point in time. The LIS communicates with the computer and the PDA on the same radio frequency.

At a logical level, both the computer and the PDA each have a runtime system (the Java based mobile agent system Mobile Spaces [12] for example). These runtime systems can run a mobile agent we call app. Moreover, app moves from the runtime system on the computer to the runtime system on the PDA when both the previously mentioned tags are within the presence of the sensor.

6.1 PDANet:Views

We model PDANet as the network type :PDANet. A views diagram of :PDANet is presented in figure 9. Here, two type diagrams are declared. These are structured into two views named Physical and Logical. These diagrams define the potential structure that instances of :PDANet may exhibit as seen from two different points of view.

Three types are declared in the header of the views diagram. These types may be used in the internal structure of all the diagrams that are declared in the views diagram. "L = :MA" means that the type L equals all instances of the component type named MA. Consequently, connectors associated with the message type L, consist of channels that may forward component instances of :MA.

6.2 PDANet:Type::Physical

The type diagram that is contained in the physical view is presented in figure 10. Here the region labeled Cell models the infrared transmission radius of the sensor. This region contains one unnamed network instance of type :Cell. The region containing comp:AH (the computer) and Tag (the tag associated with the computer) is always

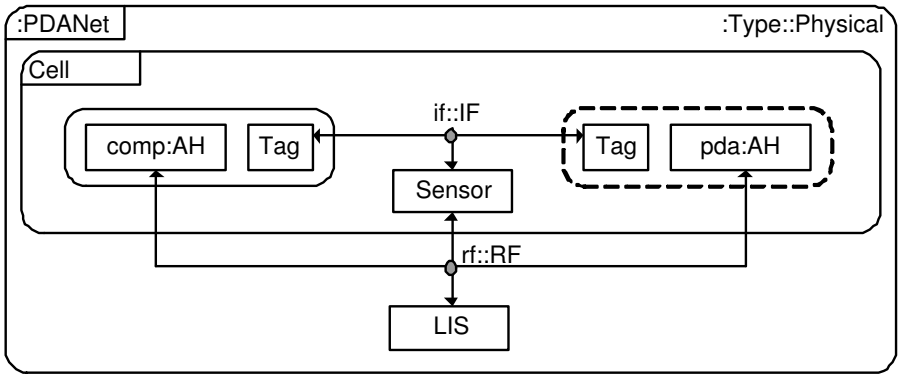

Figure 10: Type diagram of PDANet with respect to the physical view

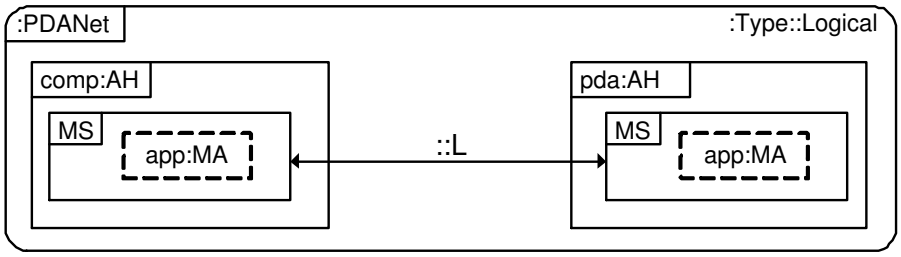

Figure 11: Type diagram of PDANet with respect to the logical view

contained within this transmission radius. This region consists of an unnamed network instance of an unnamed network type (because it is not labeled). The dynamic region containing pda:AH and Tag models the fact that the PDA and its associated tag may or may not be within the transmission radius at a given point in time. More precisely, a network n of :PDANet contains a network of type :Cell which contains an unnamed network of an unnamed network type that may or may not be part of Cell during that networks lifetime.

The connector named if of message type IF represents the infrared communication between the tags and the sensor. Similarly the connector named rf of message type RF represents the radio communication between the computers, the sensor and the LIS. Both these connectors represent connectivity at the physical level. Furthermore, both connectors are associated with a merge-split node (its graphical notation is a grey filled circle). A connector that is associated with such a node contains a single shared channel that all components it connects may forward messages to and receive messages from. E. g. the connector named if consists a single shared channel that both instances of :Tag and the instance of :Sensor may send messages to and receive messages from.

6.3 PDANet:Type::Logical

The diagram that is contained in the logical view is presented in figure 11. Here, the internal structure of the two composite component instances of :AH is specified. Specifically each instance of :AH contains an unnamed instance of MS which in turn may or may not contain app:MA at a given point in time.

The unnamed connector in figure 11 is associated with the message type L. Since this message type equals the component type MA (as defined in figure 9), instances of type MA can be sent along the channels contained in this connector. Specifically, the component instance named app of :MA (the mobile agent) may be sent from one instance of :MS to another via the channels contained in the connector labeled ::L.

7 Related Work

A good overview of graphical specification methods and techniques can be found in [16]. Notable state-of-the-art languages/methods for modeling structure are Specification and Description Language (SDL) [7], Unified Modeling Language (UML) [11], Real-time Object-Oriented Modeling (ROOM) [15], and FOCUS [3]. The relevant parts of these languages/methods for graphical structure modeling which we refer to are the constructs for structuring agents in SDL-2000, composite structures in UML 2.0, constructs for structuring actors in ROOM, and the graphical style in FOCUS.

ROOM separates between static and dynamic constructs in the specification of potential structure. The relevant dynamic constructs are *dynamic actors* and *dynamic actor relationships*. These are similar to, but less expressive than, dynamic parts and dynamic connectors, respectively. The reasons are that (1) ROOM does not enable the specification of a lower and an upper bound on multiplicity and (2) ROOM does not enable the specification of named actors (similar to named component instances). As a consequence, mobile code networks cannot be specified like it is done in diagram given in figure 11. Here app is the name of a component instance, and the two parts labeled app:MA may consist of the *same* component instance (at different points in time). This could not have been specified explicitly without named components instances.

UML does not separate between static and dynamic constructs, but the constructs called parts in UML are similar to dynamic parts in MEADOW. That is, UML does enable the specification of upper and lower bound on the multiplicity of parts, and the semantics of this is similar to the semantics of maximum and minimum cardinality of dynamic parts in MEADOW. UML does not have the equivalent of static components or dynamic connectors (connectors in UML are similar to static connectors in MEADOW). Consequently, a specification like the one in figure 3 where dynamic connectors are used, and figure 10 where static components are used, cannot be made using the constructs of UML. The individual instances that a part in UML consists of, can not be named as in MEADOW. Unlike the other languages we discuss in this section, snapshots of structure can be specified in UML.

SDL agent diagrams can be used to impose constraints on the maximum number of instances that a process set (similar to a component set) may consist of. In this way, constraints on how the configuration of a system may change over time can be captured in a structural diagram. However, a minimum bound on the cardinality of process sets cannot be specified explicitly, and SDL does not distinguish between static and dynamic constructs. Furthermore, individual instances cannot be named as in MEADOW.

The graphical style of FOCUS does not distinguish between static and dynamic constructs. Upper and lower bounds on the cardinality of components sets can not be specified. FOCUS does however, unlike the other languages we have discussed, allow individual instances of component sets to be named.

Of all the previously discussed languages, only UML enables the specification of snapshot structure. However, UML does not have constructs for relating snapshot dia-

Construct	ROOM	UML	SDL	FOCUS
Potential specification	Yes	Yes	Yes	Yes
Snapshot specification	No	Yes	No	No
Snapshot diagram relationship	No	No	No	No
Dynamic/static connector	Yes	No	No	No
Dynamic/static component	Yes	No	No	No
Upper/lower bound	No	Yes	Yes*	No
Instance naming	No	No	No	Yes

* Upper bound only

Table 1: Classification constructs for expressing dynamic reconfiguration.

grams in time. Obviously, our object-oriented and ad hoc network example cannot be specified as in the figure of section 4 and 5, respectively. Relating snapshot diagrams in time is a way to describe how networks may change over time, and this cannot be done in the structural diagrams of any of the four languages we have discussed in this section.

The discussion of this section is summarized in table 1 where the most important constructs of MEADOW for specifying dynamic configuration are listed together with information as to whether or not these constructs are supported in the four languages discussed previously.

8 Conclusions and Future Work

We have presented a language called MEADOW for specifying dynamic networks from a structural viewpoint. We have demonstrated the language in three examples addressing an object-oriented network, an ad hoc network and a mobile code network. Specification of dynamic reconfiguration is achieved through the clear distinction between *snapshot diagrams* of the structure that networks may exhibit at a point in time and *type diagrams* of the structure that networks may potentially exhibit over a period of time. Dynamic reconfiguration is modeled through (1) the construct of relating snapshot diagrams in time and (2) static/dynamic constructs that constrain the potential structure of a network. In addition, MEADOW has powerful constructs for increasing the scalability and generality of network specifications compared to the other languages listed in the previous section. See [14] for details.

The obvious advantage of increased expressablity is that diagrams can be specified at a higher granularity and thus more information can be conveyed through such diagrams. Another advantage is that constraints in structural diagrams can be used as a basis for automated model checking by checking whether the behavior of components abide to their associated constraints expressed in the structural diagrams. All of the languages we discussed in the previous section provide a rather limited basis for model checking with respect to dynamic reconfiguration.

Currently, MEADOW does only distinguish between specifications that constrain components throughout their whole lifetime (type diagrams), and specifications that constrain components at a single point in time (snapshot diagram). However, it would also be interesting to specify diagrams that apply for a *period* of time, where that period

of time is not necessarily equal a single point in time or to the lifetime of a component. This is in fact a more general way of specification since both type diagrams and snapshot diagrams would be special cases of such a "period-diagram". Such a modification of MEADOW would require the introduction of explicit timing constraints, and it is uncertain whether the advantages of this added expressability would outweigh the potentially reduced level of comprehensiblity appropriateness. A further investigation into these matters would nevertheless be interesting.

In order for MEADOW to be used for simulation purposes, it must be used in combination with a language for modeling behavior such as the π-calculus as previously pointed out, but also STATECHARTS or (certain parts of) FOCUS for example. Hence, future work on developing constructs for specifying behaviour of compoentents, or alternatively on how to combine MEADOW with existing such languages, would be interesting. Such a combination would probably result in the identification of generic *operations* than can be associated with components. Examples of such operations are send/receive message, migrate component, send/receive port, create/kill/copy component.

MEADOW does not have a formal semantic definition, nor is it supported by any computerized tool, hence future work in this direction is also a natural next step.

Acknowledgements

The research on which this document reports has partly been founded by the Research Council of Norway through the project SECURIS (152839/220).

References

[1] S. H. Bae, S.-J. Lee, W. Su, and M. Gerla. The design, implementaion, and performance evaluation of the on-demand multicast rounting protocol in multihop wireless networks. *IEEE Network*, 14(1):70–77, 2000.

[2] S. Basagni. A mobility-transparent deterministic broadcast mechanism for ad hoc networks. *IEEE/ACM Transactions on Networking*, 7(6):799–807, 1999.

[3] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems. FOCUS on streams, interface, and refinement*. Springer-Verlag, 2001.

[4] P. Clements, R. Kazman, and M. Klein. *Evaluation software architectures: methods and case studies*. Addison-Wesley, 2001.

[5] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on software engineering*, 24(5):342–361, May 1998.

[6] Radu Grosu and Ketil Stølen. Stream-based specification of mobile systems. *Formal Aspects of Computing*, 13:1–31, 2001.

[7] ITU-T. *Specification and description language (SDL), ITU-T Recommendation Z.100*, 2000.

[8] T. Korson and J. D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–90, 1990.

[9] S.-J. Lee, M. Gerla, and C.-K Toh. A simulation study of table-driven and on-demand routing protocols for mobile ad hoc networks. *IEEE Network*, 13(4):48–54, 1999.

[10] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[11] OMG. *UML 2.0 Superstructure Specification. OMG Adopted Sepcification ptc/03-08-02*. Object Management Group, 2003.

[12] I. Satoh. Mobilespaces: A framework for building adaptive distributed applications using a hierarchical mobile agentsystem. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 161–168. IEEE Computer Society, 2000.

[13] I. Satoh. Physical mobility and logical mobility in ubiquitous computing environments. In N. Suri, editor, *Proceedings of Mobile Agents: 6th International Conference (MA 2002)*, number 2535 in Lecture Notes in Computer Science, pages 186–201. Springer-Verlag, 2002.

[14] Fredrik Seehusen. Meadow - a dataflow language for modelling large and dynamic networks. Master's thesis, Department of Informatics, University of Oslo, August 2003.

[15] Bran Selic, Grath Gullekson, and Paul T. Ward. *Real-time object-oriented modeling*. Wiley, 1994.

[16] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, 1998.