CoNSoLe
A Domain Specific Language for Network Services

The PRINTYPE OLIVER
The OLIVER STANDARD VISIBLE WRITER No. 9
TAB
RIGHT
LEFT
FIG
CAP
No 9
The Oliver Typewriter Co.

Thesis for the masters degree Computer Science.
Department of Computer Science, University of Twente, the Netherlands

Author: Daniël van 't Oever

Title:	CoNSoLe: A Domain Specific Language for Network Services
Author:	Daniël van ’t Oever
Student number	s0107980
Date	May 14, 2008
MSc program:	Computer Science
Track:	Software Engineering
Institute:	University of Twente, the Netherlands
Faculty:	Electrical Engineering, Mathematics and Computer Science
Company:	TNO
Department:	Information and Communication Technology
Address	Eemsgolaan 3 9727 DW Groningen
Committee:	dr. I. Kurtev (First supervisor) University of Twente dr.ir. K.G. van den Berg University of Twente prof.dr.ir. M. Akşit University of Twente prof.dr R.J. Meijer University of Amsterdam TNO ICT, Delft

University of Twente
Enschede - The Netherlands

Preface

The start of my graduation was also the start of a new period in my life. I moved from Enschede to Groningen, from the University of Twente to the office of TNO, from my old room to my new room. I also decided to join a student union: the Navigators. The best of all these new things were all the new people that I got to know.

I would like to thank Robert Meijer, my supervisor at TNO for his patience, support and the interesting – sometimes almost philosophical – conversations. Also Rudolf Strijkers for the discussions and all the other people at TNO.

Ivan Kurtev, was my supervisor at the University of Twente. I would like to thank him for the pleasant and positive way of guiding me through the graduation process. The use of Skype and Video conferencing programs was a good way to confer our views, which makes him a rather modern teacher in my opinion. I would also like to thank Klaas van den Berg for reading my work.

Finally I am grateful for all my good friends and my family for always being there.

Groningen,
May 2008

Daniël van 't Oever

Abstract

A network provides a service to its users. Most modern networks are based upon the TCP/IP protocol. The main service provided by this protocol are best-effort, end-to-end connections. The behavior of this protocol is standardized and there are no default mechanisms to change the default behavior or to introduce new behavior. to improve this situation, programmable networks were developed, that allow the user to *program* how the network should behave. Examples are Active Networks and User Programmable Virtualized Networks (UPVN). This opens the door for application-specific network services; an application can optimize the service provided by the network for its own use.

With a General Purpose Language (GPL), every technically possible network service can be programmed. But since the language is *general*, it requires every detail to be specified. On the other hand there are Domain Specific Languages (DSL) that address problems in a small domain. Because the domain is known, a solution for a problem in that domain can be short and efficient. A drawback is that problems that are only slightly outside of the domain can not be solved by the DSL.

In this research we want to reveal the mechanisms or statements that are common for creating network services. We do this by creating a network service programming language. We chose to develop a DSL, since every addition to a DSL can be considered domain knowledge. This domain knowledge could be mixed up with knowledge from other domains if we would develop a GPL. We only investigate network services that require the cooperation of *multiple* network elements. Network services that can be provided a single Network Element (NE) are not considered.

We selected five case studies that introduced a list of problems that the DSL was expected to solve. Each case study introduced new statements in the DSL or reused statements from other case studies. When all case studies were solved, a domain analysis was performed from which a domain model was built. This domain model is reflected in the meta-model of the DSL.

We conclude that the main activities of a network service DSL are the retrieval of information from the network, the management of the topology of the network, the management of routes in the network (manage packet processing) and to support third-party applications with the distribution and deployment in the network. The latter is done because network services easily enter another domain in which the DSL can not be of much assistance. Therefore we included support for third party applications to support them as much as possible.

Compiling a network service specification can be difficult if it depends on the topology of the network, for example a shortest path between two NEs. The appearance of new NEs can cause the shortest path to become invalid. We developed a strategy in which topology related feedback from the network is used as a trigger to recompile the network service specification, hereby repairing the possibly invalid network service.

Parts of this thesis will be published as:

Robert J. Meijer, Daniël van 't Oever, Rudolf Strijkers, Ivan Kurtev, "Creating Network Services with a Domain Specific Language".

Table of Contents

List of Abbreviations

AC	Application Component
ATL	Atlas Transformation Language
ATL	Atlas Transformation Language
CNS	Collective Network Service
DSL	Domain Specific Language
EMOF	Essential MOF
GPL	General Purpose Language
HQL	Hibernate Query Language
KM3	Kernel meta-meta-model
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta-Object Facility
NC	Network Component
NE	Network Element
OMG	Object Management Group
OSPF	Open Shortest Path First
SQL	Structured Query Language
TCP/IP	Transport Control Protocol/Internet Protocol
TCS	Textual Concrete Syntax
UPVN	User Programmable Virtualized Networks
VM	Virtual Machine

List of Tables

List of Figures

Chapter 1

Introduction

Consistency is the enemy of enterprise, just as symmetry is the enemy of art

George Bernard Shaw

1.1 Background

As the majority of internet routers are unaware of the service demands of applications that send their data through them, applications are restricted to the service that is offered by the network. Many networks of which the internet is the biggest example, are built upon the Transport Control Protocol/Internet Protocol (TCP/IP) . The service provided by the TCP/IP protocol are best-effort, end-to-end connections. For most applications this service is good enough.

Adapting and introducing network services is an activity primarily for network operators such as Internet Service Providers. A network operator can only optimize the network when the service demands of the end users are understood and when the operator can react in time.

Currently, applications have very limited possibilities of adapting a network service themselves. One of the reasons is that the socket interface – the most popular interface to services in TCP/IP based networks – is limited by design. A socket offers end-to-end connections which are abstracted as a file with support for read and write operations, hereby hiding the underlying details of the connection. Also, most internet routers are configured in such a way that they collectively provide a best effort shortest path routing service.

In sensor networks for example, applications that transfer data over the network need to take information about the battery lifetime and the topology of the sensor nodes into account. A socket interface does not expose this information. To satisfy the demand of such applications nonetheless, alternative protocols are created or amendments to existing protocols are proposed. Indeed, it is not possible to make application specific optimizations in TCP/IP based networks.

Alternative programmable network models such as Active Networks [35] are developed to provide applications with the mechanisms to adapt or introduce network services. Now that programmable networks have shifted the attention from *who* programs the network to *how* to program the network, the research community started developing programming languages that are optimized for the domain of network services [12][28][20]. These languages are either General Purpose Languages (GPL) and require much effort to solve any particular problem or they are Domain Specific Languages (DSL) which are more efficient, but can only solve problems in a small domain.

Further efforts are required to investigate the constructs that are common for network services and what is common in programming network services so reusable solutions can be created.

1.2 Problem Statement

The focus of this thesis is on introducing network services rather than using or adapting existing network services. Introducing network services is no new activity, this can already be done with a GPL. But since every technically possible problem can be solved with a GPL, there is no clear notion of the mechanisms that are required to introduce network services. It is expected that by developing a DSL, it will become clear which mechanisms are essential or common to introduce network services.

1.3 Research Questions

The question that will be investigated in this research is: *"Can the design of a domain specific language reveal the constructs that are common to network services?"*. This research question is further refined with the following sub questions:

What is programming a network service? There is no clear definition of the term 'network service'. A clear definition is essential before a language can be designed that can specify network services.

What is the architecture of a network service language? With the architecture we mean the concepts that are included in the language, the syntax, the semantics, the runtime environment and the compiler.

What are common instructions to compose network services? Just like any computer program can be reduced to a few assembler instructions, we study if there are instructions that are common or even essential for network services.

Is there an instruction set that can support all network services? If there happens to be a common set of instructions for composing network services, is it complete?

How to cope with the dynamics of the network? In networks, topologies change, traffic varies, and so does the content that travels over the network. Since the compilation target of the DSL is the network, there is a dynamic compilation target. How can applications remain valid if the target changes?

How can DSL design help understand the concepts of the application domain? Designing a new language requires a good understanding of the problem domain

[22]. Since these findings are explicitly recorded in artifacts like the language and supporting tools, the concepts of the application domain should be recognizable in these artifacts.

1.4 Approach

To program the service of a network, we had to choose between two strategies: enrich the instruction set of a General Purpose Language (GPL) with network specific instructions or to design a new language. The risk of taking the first approach is that it is likely to start thinking in the terms and principles that come with the existing language. The second approach was taken and this allows to carefully select the domain concepts that will make up the language without getting distracted by existing programming principles. This approach also ensures that only explicit domain knowledge is added to the language, the compiler or the runtime environment. This knowledge could be lost if the first strategy was adopted.

To deal with the problem of finding the right domain concepts, we adopted the following strategy: (1) Only the absolute minimum of statements necessary to program a network service are included in the DSL. (2) Everything else that does not belong to this domain should be programmed using an existing GPL. A consequence of this decision is that not every network service can be expressed with the DSL. A GPL can complement this limited functionality of the DSL. The runtime environment in which the (compiled) DSL programs run, was designed using the same philosophy: no statements are implemented unless they are absolutely necessary. This approach should help discovering the minimal set of statements that are essential or common to network services. The DSL will be implemented using Model Driven Engineering techniques (MDE) [16], which allows us to capture the domain knowledge explicitly into models and meta-models.

To support the analysis and design process in this research, five case studies were selected, each with an application that has a different demand of the network's service. Also, analysis is based on the past work and experience of domain experts; the authors of [21].

1.5 Contributions

This research provides a contribution to the domain of DSL engineering as well as to the domain of Network engineering. Besides giving a definition of the term 'Network Service', this thesis provides the following contributions:

1. *A language whose statements are common for network services*

By creating a language that was created to solve the problems presented in five case studies, a list of statements was revealed that is common to network services.

2. *An (example) architecture for compilers that must deal with dynamic targets*

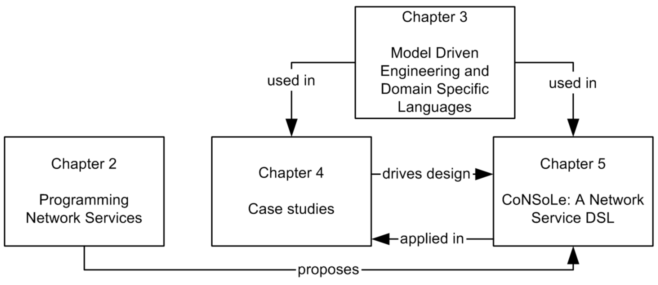
Chapter 3
Model Driven Engineering and Domain Specific Languages
used in
used in
Chapter 2
Programming Network Services
Chapter 4
Case studies
drives design
applied in
Chapter 5
CoNSoLe: A Network Service DSL
proposes

Figure 1.1: *Thesis outline*

The combination of the network domain with the domain of language design yielded an interesting case: compiling applications to a dynamic target. This is different from traditional compilers that compile to a target that is static. Although we used a relatively simple solution of recompilation to handle the problem, further research can provide more insights that can contribute to more robust applications in the distributed computing domain.

3. A virtual machine that reflects the primitive statements that are needed to create network services

Because we developed a DSL, only the statements that are essential to the problem domain (network services) end up in the virtual machine. This allows for careful investigation of these statements.

4. Employment of a DSL to solve relevant case studies for the network domain

The DSL developed in this research is used to solve case studies that go beyond the typical toy examples of programming a robot with a DSL.

1.6 Thesis outline

Figure 1.1 is an outline of the structure of this thesis. Chapter 2 gives a definition of a network service and reviews existing programming languages targeted at the network domain as well as the network types on which they operate. This chapter also gives reasons for the design of a new DSL. Chapter 3 introduces Model Driven Engineering and its relation to Domain Specific Languages. The tools and techniques that support DSL development and the techniques that were used in this research, are explained as well. Chapter 4 describes the case studies that were selected and implemented to drive the domain analysis. From the domain analysis, a domain model is derived. Chapter 5 takes the domain model and incorporates this in the meta-model of the language. It presents the developed domain specific language, the architecture of the Virtual Machine (VM) in which the compiled DSL executes, as well as the instructions of the VM. In the end of this chapter, the compiler is presented. Chapter 6 concludes this thesis by discussing the presented work and by answering the research questions.

Chapter 2

Programming Network Services

Everything is a file

UNIX philosophy

This chapter gives an overview of the programming of network services. First the definition of a network service will be given in section 2.1. Some networks, like the internet, require special privileges to change their behavior and therefore introducing new network services is an exclusive activity. There are alternative kind of networks that allow the end-user to program its behavior and therefore allow for easier introduction of network services. The different types of networks will be presented in section 2.2. In section 2.3 several languages are discussed that are created to use, adapt or introduce network services. Section 2.4 gives rationale for developing a new language to program network services. Section 2.5 concludes this chapter.

2.1 The notion of a network service

In section 2.1.1 the concept of a network service is defined. Section 2.1.2 presents an overview of the programming languages that already exist to program network services. Each of these languages has a different purpose and is presented in more detail in section 2.3. Some languages are only created to obtain information from the network, whilst others are more general and targeted at a wider domain. Table 2.1 gives an overview of these languages and groups them by purpose.

2.1.1 Definition of network service

A service is an act of work to support another. In a network, a service can support an end-user or another network service. Typical services found in current networks are information exchange, authentication and storage. Some of these services require the cooperation of multiple NEs, whilst other services can simply be provided by an individual NE (e.g. storing a file). Since we want to investigate the mechanisms of *network* services, we will only focus on services that can only be provided by multiple NEs and we do not investigate the mechanisms for programming individual NEs. From now on, the following definition will be used:

A Collective Network Service (CNS) is an act of work that requires the participation of applications on multiple NEs, that in concert produce a service for the end-user.

2.1.2 Programming a network service

Programming languages for network services support a developer in different ways. Table 2.1 gives an overview of the different purposes of a programming language when it comes to network services.

Language	Language purpose	Description
NML (section 2.3.1), TinyDB (section 2.3.3)	Service utilization	Utilize existing services in the network e.g. obtain information, upload a file or perform distributed calculations.
NML (section 2.3.1)	Service adaptation	Change existing network behavior e.g. changing routing rules or increasing the maximum size of an email message
PLAN (section 2.3.2), MOB (section 2.3.4)	Service introduction	Introduce new functionality on individual NEs that will produce a new network service e.g. new protocols

Table 2.1: *Different network service aspects and how they are addressed by various existing languages, ranging from domain specific to general purpose.*

Network service *utilization* is an activity that uses existing network services in a read-only fashion; the behavior of the service is not modified. Using existing services is done mostly by end-users and is the main activity in current networks of which the internet is the biggest example.

Network service *adaptation* is an activity that changes the behavior of a network service and requires the right permissions to do so. A programming language – possibly as simple as being a configuration file – is used to change the behavior of the network service. This activity is primarily for users in the domain of network maintainers such as Internet Service Providers.

Finally, programming languages are used to create new networks services, either by combining existing functionality or by completely creating the new functionality from scratch. From now on we call this programming activity network service *introduction*.

2.2 Network models

This section presents three network types and how well they support the network management activities outlined in table 2.1. It is difficult to adapt or introduce new network services in IP-based networks. IP-based networks are discussed in section 2.2.1. As a reaction, alternative network models are developed, of which two models are discussed in section 2.2.2 and 2.2.3.

2.2.1 IP-based Networks

Currently, most networks are based on the IP-protocol of which the internet is the best known example. Inherent to this network model are the characteristics of its connections, which are end-to-end connections over which the data is transported in a best-effort fashion. All the services provided by IP-based networks are built upon this kind of connections. Changing the best-effort behavior of the connections requires drastic changes to the IP-protocol. The process of getting new protocols introduced or existing protocols adapted can take several years. For example, it took 21 year before the TCP/IP protocol became standardized [30].

In IP-based networks there are no default mechanisms to let anyone but the network administrator introduce or change the behavior of the network. As a consequence all the services in this kind of network are based upon best-effort end-to-end connections.

2.2.2 Active Networks

The Active Networks [35] approach breaks open the predetermined behavior of IP-based networks by making routers programmable. A router can be programmed by executing program code that can be encapsulated in the packets that travel over the router or via some off line method. Both methods allow the introduction of new routing services that can deliver more than best-effort end-to-end connections. Active Networks is a step forward toward application-specific routing and with this also toward application-specific network services.

2.2.3 User Programmable Virtualized Networks

User Programmable Virtualized Networks (UPVN) [21] is a conceptual programmable network model that allows developers to interact with network elements by providing them with a software handle – or proxy – to a Network Element (NE). This handle is called a Network Component (NC) . An application that embeds a NC gains access to the resources of the corresponding NE, opening the door for applications to add new functionality; the network is virtualized *in* the application. This is illustrated by figure 2.1. A piece of software that was put on a NE in favor of the application is called an Application Component (AC) . Applications that embed multiple NCs can now compose new network services by deploying and facilitating ACs that in concert produce a network service. Now applications can not only program the network layer (routing) of a NE – as is the main focus of Active Networks – but applications can also program every other aspect of a NE that is exposed to the application.

2.3 Existing network service programming languages

This section gives an overview of domain specific and general purpose programming languages that are created for the network domain. Languages developed for standard IP-based network models are usually created to *use* existing network services. These

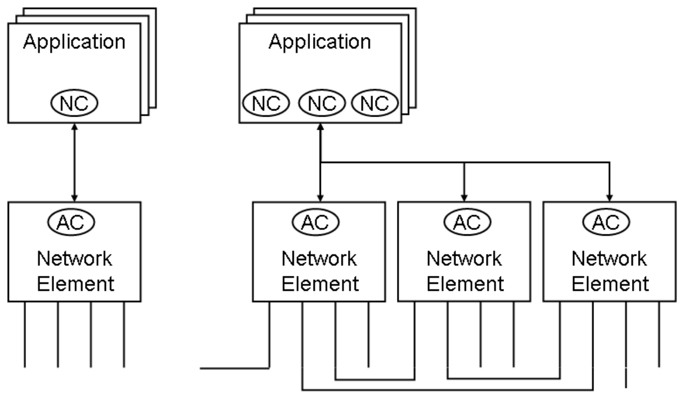

Figure 2.1: *The UPVN model in which the network elements (NEs) are virtualized inside the application. This can be done by embedding one or more network components (NCs) which are the software interfaces to the NE.*

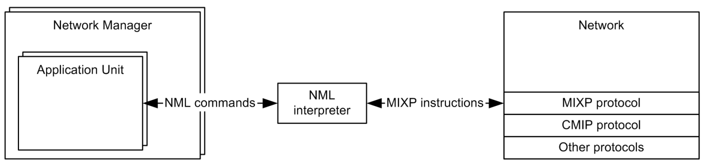

Figure 2.2: *One or more application units (AU) form a Network Manager. AUs issue NML statements that get compiled to low level protocol instructions*

are described in section 2.3.1. The languages that are created for alternative network models, or modified IP-based networks also facilitate the adaptation or introduction of network services. These are discussed in section 2.3.2, 2.3.3 and 2.3.4.

2.3.1 Network Management Language(NML)

In [36] a Network Management Language (NML) is proposed and is used as an intermediate language between network management applications and network management protocols[1], this is illustrated by figure 2.2. This DSL provides mechanisms for both *monitoring* and *controlling* the network. Network management applications are called *Application Units* (AU) that provide an interface between the network and the (human) administrator. An AU can for example display a topographical map of the network by issuing NML commands to the NML interpreter.

The interpreter translates high-level NML commands to a collection of low-level instructions for the Management Information Exchange Protocol (MIXP). This protocol is used to query management data of a NE using *get* and *set* operations. This protocol is no longer in use, but it can be compared to the Simple Network Management Protocol

[1]A protocol can be regarded a domain specific language that defines the rules governing the syntax, semantics, and synchronization of communication

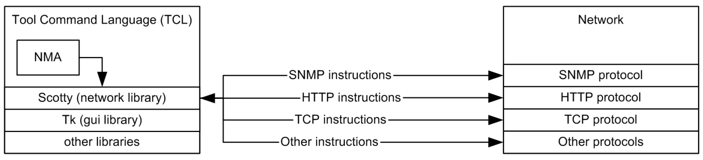

Figure 2.3: *Scotty is a domain specific library. Network Management Applications (NMA) use Scotty high-level functions that 'compile' to low level protocol instructions.*

(SNMP) [5] that is still widely being used in IP-based networks.

A similar language is presented in the work of [32] named Scotty. This language compiles to instructions for a collection of protocols than can be used to manage and obtain information about the network. Scotty is an extension to the Tool Command Language (TCL) and is more a network management library than a language. Supported protocols are HyperText Transfer Protocol (HTTP), Domain Name Server (DNS), netdb (to query local network databases) and SNMP. Figure 2.2 outlines the architecture of Scotty.

2.3.2 Packet Language for Active Networks (PLAN)

This language is built upon the concept of Active Networking, which addresses the problem that current networks are hard to change, by defining a programming interface. New functionality can be introduced by wrapping programs into packets, called *active packets* that are executed on every NE that processes the packet. Functionality can also be added by downloadable router extensions called *switchlets* [12].

The PLAN language was designed to perform functions such as network diagnostics, network service management and configuration as well as distributed and programmable communication between applications and network elements. It is not a general purpose language, as general purpose expressibility can be provided by switchlets. The language is based upon the simple typed lambda calculus and contains a subset of the features found in common functional programming languages. PLAN contains some domain specific statements such as **OnRemote** which allows active packets to be sent to a given destination and **defaultRoute** which specifies to use the default IP routing policy.

Figure 2.4 is an example of a PLAN application that represent the well known *traceroute* application. The application sends itself as an active packet over the network. In case a NE does not contain the necessary extension, it can be dynamically downloaded from the previous NE. In case of the traceroute program no special extensions are needed.

2.3.3 TinyDB

TinyDB [20] abstracts the network as a virtual database. NEs have sensors attached to them that can be queried. For this purpose the Structured Query Language (SQL) has been extended with timing conditions that allow the user to express the duration and the interval of a query. It frees the developer from having to write complicated C programs that must be distributed on all the NEs in the network. Queries are optimized by power-efficient in-network processing algorithms. Queries that result in a true or false

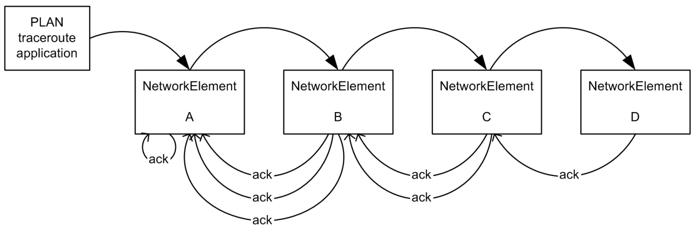

Figure 2.4: *A PLAN application is put on the network as an Active Packet. On every NE the Active Packet is executed which results in the sending of an 'ack' packet to the start NE as well as the sending of itself as an Active Packet to other NEs.*

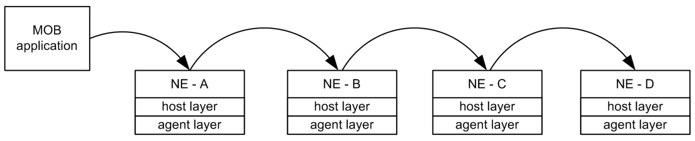

Figure 2.5: *A MOB application contains instructions to will cause the host layer to serialize the application and send it to another NE. The agent layer implements the services that are used by MOB applications. Services itself are MOB applications as well.*

value can be used to trigger a signal that will be fired over the network. To transport the requested information, the operating system on the NEs uses an ad-hoc routing protocol that forms a spanning tree. Information flows to the root of the spanning tree (the sink) where it can be collected and processed by a computer.

2.3.4 MOB

MOB is a high-level language build upon the π-calculus [24]. Its purpose is to provide a formal framework for network agents. The MOB language is best used for formal experiments that need constructs that are at a higher level than the constructs provided by the π-calculus itself. Figure 2.5 illustrates the MOB architecture.

Agents and services are the primary building blocks. Services are always implemented by agents. Agents thus provide and require services. Agents are multithreaded, supported by the *join* and *fork* instruction. Moving an agent is implemented by the *go* instruction. Incorporation of external applications is done by an *exec* instruction. Most likely this instruction assumes some default communication mechanism with an external application. It is not specified how this is implemented.

2.4 Rationale for a new language

A DSL can be used to solve problems that occur over and over again in a particular domain (e.g. routing). There are aspects that are the same every time the problem occurs. If these are the only aspects, a solution can be created once and for all and no

DSL is needed. When there are aspects that are different everytime the problem occurs, the solution needs to be adapted. A DSL allows a programmer to express the parts of an existing solution that needs adaptation.

A network service can be regarded a solution that may need adaptation in certain situations. All the DSLs that are described in this chapter allow the user to fill in the variable part of a solution: a network service. The desire for new network services is reflected in the large number of proposed languages for networks [19][26][12][20][28]. Sensor networks have pushed the desire for new network services, by having more extreme demands on the service of a network (e.g. access to the battery lifetime of a NE). These services can not be easily implemented or provided by IP-based networks, hence alternative network types (Active Networks, UPVN) and languages for these networks are being developed.

If we assume that every network element can be programmed, it is obvious that a GPL can be used to program every technically possible network service. This also comes at the expense of having to program every detail of the problem. A DSL improves the *ease of use* to specify solutions for problems in the domain for which it was developed. Although a GPL can solve the same problem, it requires significantly more effort and technical knowledge of the domain compared to using a DSL for the same task.

Instead of developing a solution for a particular network problem, we want to reveal the *mechanisms* that are common for network services. In the languages described in the previous sections, we have seen that these mechanisms appear in both DSLs and GPLs, however it is not clear for what exact purpose these mechanisms are included. In these languages, solving the problem is superior to the mechanisms that contribute to the solution. In this thesis we take an opposite approach and instead of solving a particular problem, we select five different cases, each with different problems, and from there we develop a DSL. By doing this, we focus on the mechanisms common to all the different problems, rather than solving a particular problem.

2.5 Conclusions

New network services require changes to the NEs as we currently know them. To achieve this, alternative network models such as Active Networks and UPVN are being developed. Most current network DSLs are focused on making the use of one *specific* network service easier. Current GPLs specially created for networks, are similar to 'normal' GPLs but are enhanced with dedicated constructs such as built-in distribution mechanisms or formal properties. Since the current network programming languages are more focused on particular problems rather than on the underlying concept of network services, a new language will be developed to investigate the mechanisms that are common for network services.

Chapter 3

Model Driven Engineering and Domain Specific Languages

Nothing is lost, nothing is created, all is transformed

Antoine-Laurent de Lavoisier

This chapter gives on overview of the concepts that are used to implement the language and the compiler. First explain what we consider a model in section 3.1, then we explain the notion of Model Driven Engineering (MDE) and Model Driven Architecture (MDA) in section 3.2. Then we explain what a meta-model is in section 3.3. Then the concept of a *model transformation*, which is a basic operation in MDE, is explained in section 3.4. Section 3.5 is devoted to domain specific languages. Now that these definitions are clear, we show the overlap between Domain Specific Language development and Model Driven Engineering in section 3.6. Section 3.7 concludes this chapter.

3.1 Models

There are many definitions of the term *model*. In this thesis we adapt the following definition by Kleppe et al [17]:

"A model is a description of a system written in a well-defined language"

This definition underlines the relation between a model and a system. Figure 3.1 illustrates this relation. The figure is called the *DDI account*: Denotation, Demonstration, Interpretation and is introduced by Hughes[13]. Elements of the system are *denotated* by elements of the model. A model is expressed using a modeling language. The process of asking questions to the model is called *demonstration* and happens "entirely within the model". Finally the answer to a question is *interpreted* in the context of the system. For example, the answer to a question to the database (model) about the salary of the youngest employee is related back to a real person (system).

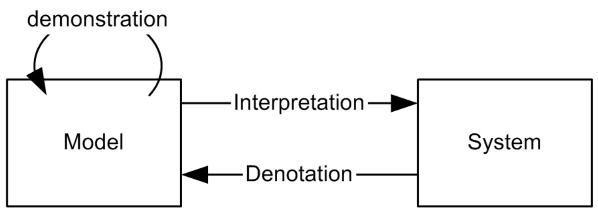

Figure 3.1: *The relations between a system and a model according to [13] The figure is called DDI account: Denotation, Demonstration and Interpretation.*

3.2 Model Driven Engineering

Model Driven Engineering (MDE) [16] is a software development paradigm that considers models as the primary building blocks of software engineering. MDE is a term that unifies all development approaches that use models as the primary form of expression. The Model Driven Architecture (MDA) [23] approach, proposed by the Object Management Group (OMG) was an initial idea from which MDE practices evolved.

MDA tries to abstract from any particular technology by the use of models. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification [23].

System development by MDA is done by creating a Platform Independent Model (PIM), which is “a view of a system from a platform-independent viewpoint”. Then this PIM is transformed to one or more Platform Specific Models (PSM) which is “a view of a system from a platform-specific viewpoint”. Finally the PSM is transformed to code. The transformation operation is implemented by a *model transformation.* Model transformations are explained in more detail in section 3.4.

3.3 Meta-models

There are models that define the allowed elements of another class of models. Such a model is called a meta-model. A good definition is given by Seidewitz [33]:

“a meta-model is a model of models expressed in a given modeling language”

So a meta-model is also a model, expressed in a (well-defined) language and contains the concepts and rules that determine the set of possible models that can be denotated by the meta-model.

Figure 3.2 shows that a model can reside at different levels. The different levels are numbered. At the bottom of the stack, model-level m1, are models that are specified by a modeling language, possibly a meta-model residing one level higher in the stack at meta-model level m2. A meta-model can also be an instance of another model: a meta-meta-model. Meta-meta-models reside at the top of the stack at meta-meta-model level m3. Although it is possible to expand this stack even further, this is usually not necessary since most meta-meta-models can be expressed in terms of themselves.

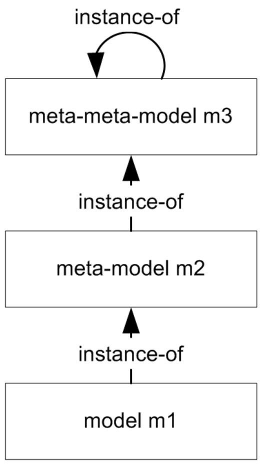

Figure 3.2: *The meta-modeling stack.*

If we subject a meta-model to the DDI account of Hughes, what is the system that is denotated by the meta-model? Clearly the system is the collection of every possible model that can be expressed by the meta-model. Figure 3.3 is an example of systems and models at different modeling levels. The weather system is represented by two different models at model-level m1 in the same language, the employees system is represented by a similar model in different languages. The Java language meta-model and the C++ language meta-model are at model-level m2 and have the same meta-meta-model: the Extended Backus-Naur Form (EBNF) model, at model-level m3. This model can be expressed in itself and is used to formally and unambiguously specify the grammar – or allowed sentences – of a language.

ECore models

Modeling in MDA is done on models which are an instance of the Meta-Object Facility (MOF). Just like EBNF plays the role for defining programming language grammars, MOF plays this role for defining meta-models. A popular implementation of MOF is ECore. Actually, ECore is an implementation of a subset of MOF, called Essential MOF – which is a subset of MOF 2.0. – and is targeted at Java development. The ECore meta-meta-model is the core model for applications developed for the Eclipse Modeling Framework (EMF) [4]. In this research we will use ECore models for defining the meta-models of the DSL and for the compiled DSL.

3.4 Model Transformations

As stated before, in MDA a model transformation is the basic operation to go from one model to another model. In this thesis we will use the following definition by Kleppe et al. [17]:

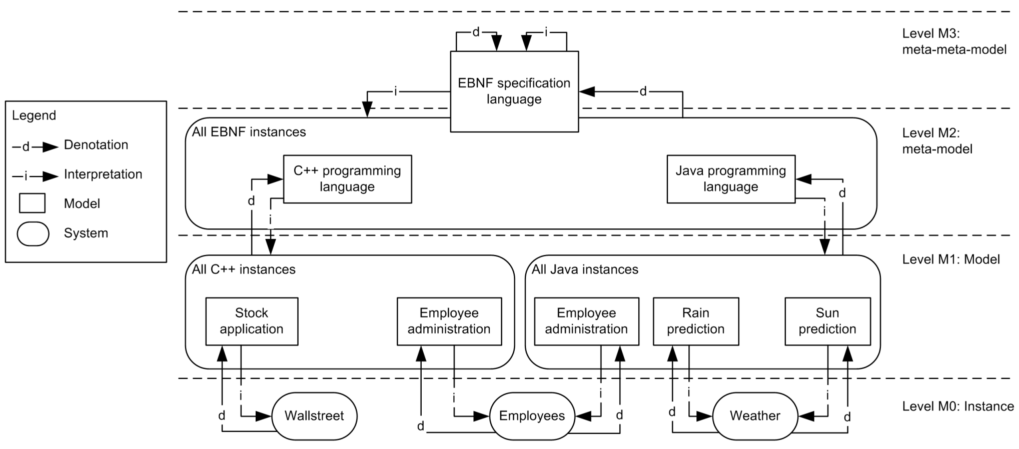

Figure 3.3: *An example of the relations between models and systems at the various model levels.)*

"The automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language"

Figure 3.4 shows the model transformation pattern that illustrates the above definition. The transformation takes as input both the meta-model of the source language and the meta-model of the target language. The transformation is written in a transformation language and specifies *how* elements of meta-model A should be transformation to elements of meta-model B. Now the transformation can take a model A that conforms to meta-model A and according to the transformation specification, this model will be transformed to a model B that conforms to Meta-model B.

Atlas Transformation Language (ATL)

An implementation of the concept of a model transformation is the Atlas Transformation Language (ATL) [2]. In ATL it is possible to transform an ECore model into another ECore model, provided that their meta-models are given as well. An ATL transformation definition describes (either declaratively or imperatively) how elements in the source meta-model should be created – and possible manipulated – in the target model. In this thesis we will use ATL to implement the compiler. For more details on how this is implemented, see section 5.6.6.

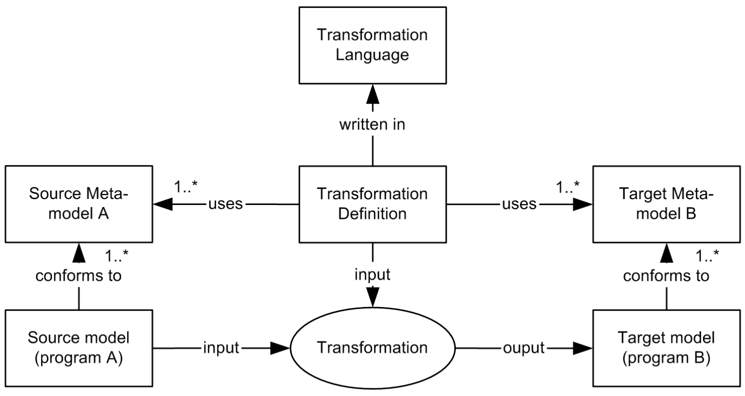

Figure 3.4: *The Model Transformation Pattern.*

3.5 Domain Specific Languages

The primary activity of this research is the development of a domain specific language. We will use the following definition by Deursen et al. [8]:

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

In this thesis, the domain is indeed restricted to network services. Finding the appropriate expressive power is a trade-off that every language designer has to make, consider the following quote of Alan J. Perlis: *"Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy"* A Turing tar-pit [29] is the place where a programming language has become so minimal and general that any program can be created, but writing any specific program has become a very difficult and user unfriendly task. This underlines the trade-off between a generality and specificness; the wider the class of problems that must be solved by a language, the less efficient the language is for solving any particular problem. On the other hand, the smaller the class of problems, the more efficient a language can tackle the problem, but the language is of no use for anything else. So a decision must be made about the size of the problem domain that will be targeted by the DSL.

A DSL and its compiler or interpreter can be seen as a big parametrization mechanism. Consider a DSL for the relatively small domain of string formatting. The DSL provides the mechanisms to identify substrings and mechanisms to make string conversions. The user that enters a formatting definition is filling in parameters in these mechanisms. Finally the compiler is often used to fill in platform specific parameters, e.g. locale-specific information such as date and currency formatting.

3.6 MDE based DSL development

Both MDE and DSL engineering share the idea that language engineering may help in domain modeling, in fact there is a convergence of MDE and DSL engineering [18]. Another definition in [18] states: "A DSL is a set of coordinated models". The following similarities are indicated: programs in a DSL represent a state of affairs in the domain for which the DSL has been developed, i.e. such a program is a model. A DSL that has an explicit representation of all the possible state of affairs in this domain it said to have a *meta-model*. Furthermore, the meta-model of both a DSL and a model can have different notations, for example a boxes and lines diagram or an XML document. Finally, a DSL may have an execution semantics definition, captured in a transformation to another language that already has a precise execution definition. E.g. a DSL can be transformed to Java program code.

The definition of a DSL by Kleppe [17] states that a model is written in a well-defined language, for example a DSL. So if we consider a DSL as a model, the principles and tools provided by MDE can be applied. For example we can create DSL by defining a meta-model in ECore that defines the valid relations between model elements. An instance of this meta-model represents a program written in this DSL. Assume we also have an ECore meta-model of the Java language. Now a compiler can be implemented using a model transformation. This transformation specifies how elements in the DSL program are mapped to elements in a Java model. Now the Java compiler can be used to transform this model into bytecode that can be executed by the Java Virtual Machine.

Abstract syntax and Concrete syntax

To store a model on a computer, a binary or textual notation is needed. This can also be the directly editable format. For example, ECore models are usually stored as XML files, but they can be edited in many different representations, ranging from various textual notations to graphical notations. All these notations describe the same model elements and their relations. The notation that describes the allows model elements as well as their relations is called the *abstract sytnax*. The notations that can be used to describe these elements and relations is called the *concrete syntax*.

KM3

A domain-specific language for specifying abstract meta-models is the kernel meta-meta-model (KM3) [14]. It is intended to be a lightweight textual language to define meta-models. Meta-models expressed with KM3 can be easily converted – or transformed – to a concrete model having a different notation, for example an ECore model.

TCS

The Textual Concrete Syntax (TCS)[15] domain specific language was designed to rapidly attach a concrete (textual) syntax to meta-models (expressed by KM3). From the TCS mapping, a parser can be generated automatically. This parser accepts all textual instances of the meta-model and parses this to a model with a different representation. This process is called *injection* and is illustrated in figure 3.5. At the moment,

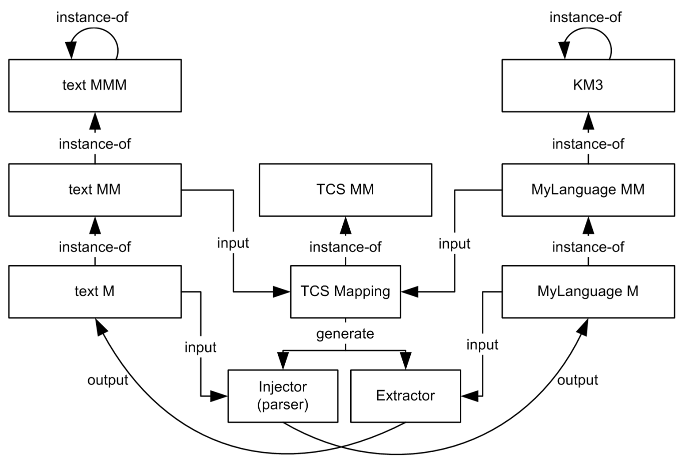

Figure 3.5: *The TCS mapping between the MyLanguage meta-model and a textual meta-model. The TCS meta-model itself is also an instance-of the KM3 model.*

the textual meta-model is used as the grammar definition for the parser-generator application named ANTLR [27]. Because the TCS mapping is bi-directional this process can also go the other way around and then it is called *extraction*.

3.7 Conclusions

In this chapter we have seen that a model can be used to represent a system with a certain purpose. Furthermore there are meta-models which are models of a class of models that conform to this meta-model. Models and DSLs have many similarities. In fact a model can represent the abstract syntax as well as the concrete syntax of a language. Therefore it is possible to apply techniques and principles from the field of MDE to these models. Model transformations are the basic operations that are used to go from one model to another.

Chapter 4

Case studies

Things should be made as simple as possible – but no simpler

Albert Einstein

This chapter presents five case studies that are implemented on a UPVN network. The case studies serve two purposes, (1) serve as input to the analysis and design process of the language presented in chapter 5 and (2) the language that resulted from the case studies was directly applied in the prototypes that implement the case studies.

The studies led to a DSL that covers four areas that are of importance for a network service language: (1) obtaining information from the network, (2) managing topology, (3) performing application specific routing and (4) supporting distributed applications. This is reflected in table 4.1 which also shows which case study contributes to which area. The case studies are described in section 4.2 to section 4.6. Section 4.7 analyses the case studies. From this analysis, a domain model is derived which is presented in section 4.7.1. This domain model will be used to define the meta-model of the language presented in the next chapter. Section 4.8 concludes this chapter.

4.1 Introduction

The first four case studies all have a different demand of the network's service. The fifth case study investigates the issues that arise when there is a dynamic compilation target (the network). All case studies have in common that the network service can only be provided by the network as a whole, and not by a single NE. As a consequence, the DSL that specifies network services contains no mechanisms that allow for the manipulation of individual NEs, but only networks as a whole or subnetworks.

Individual NEs contain facilities that ACs use to contribute to the global network service. To investigate the essential constructs for deploying network services, these facilities are implemented as instructions in a Virtual Machine (VM). This VM is described in section 5.4 and is used by the prototypes for the case studies described in this chapter.

Title	Description	Area
Topology changing applications (section 4.2)	An application creates and removes links between NEs that satisfy a given condition.	Network topology
No destination address (section 4.3)	This case deals with NEs (e.g. sensors) that transmit data without a destination address. A network service assists in the collection of this unaddressed data	Application specific routing
Network File system (section 4.4)	Distribute a file or other data in pieces over NEs. NEs multicast data to neighbors. The pieces are always in transition, hence the file is in the network. A network services assists to distribute and collect these files.	Application specific routing
Self positioning code (section 4.5)	In this case an application uses the DSL to move itself to another NE. The quality of this service depends on ability of the application and its runtime environment to move a running application.	Managing applications
Feedback driven recompilation (section 4.6)	In case the network model changes, a CNS may no longer be valid. This case investigates feedback as a mechanism to detect and react to changes in the network model.	Dealing with the dynamics of the network

Table 4.1: *Overview of the case studies that are presented in this chapter, as well as the area to which the case study contributes.*

For every case study, a part of the domain model is identified and outlined in a model. We also identify candidate instructions for the language. In section 4.7.1 all these separate domain model parts will be assembled into a complete domain model as well as a summary of the candidate instructions.

4.2 Case 1: Topology changing applications

At the moment there are few network applications that depend on the topology of the network. Having wireless connections between NEs opens the door to programmatically determine which NEs should be connected to each other. A network can use simpler routing protocols for example if its topology is a tree structure and therefore contains no loops. Also, a network may contain links whose disconnection may lead to separation of the network. In graph theory such a link is called a *bridge*. A network language should assist in finding this bridge and provide a way to remove the bridge from the network. This can be done by creating additional, redundant connections

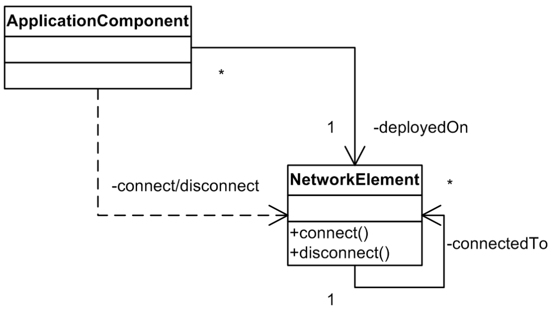

Figure 4.1: *Case 1: An ApplicationComponent can instruct Network Elements to create or remove connections with other Network Elements.*

4.2.1 Topology management as a CNS

There are only two instructions needed to manipulate a topology of a network. This can be done with a *connect(A, B)* and a *disconnect(A, B)* instruction, performed by an AC, where A and B are different NEs. These are the most primitive instructions that can provide a solution for topology management and are used as the basis for a more high-level instruction. A possible high-level topology instruction can take a graph and use it as a prescription for a desired topology. This requires the introduction of graph concepts into the language. Since this complicates the language, we decided to implement an instruction that is still at a higher level than a single (dis)connect(A, B), but does not require the notion of a graph:

```
CREATE Link FROM <select> TO <select>

DELETE Link FROM <select> TO <select>
```

With the above instructions, new connections can be easily created between collections of NEs that satisfy the constraints in the *select* clause. This clause obtains a collection of NEs and is explained in more detail in section 4.3.1. The above code is compiled to an AC that issues the low-level instructions to the NEs. This is illustrated in figure 4.1. We deviate from the standard UML class-diagram notation here by using a dashed line to indicate a method invocation.

4.3 Case 2: No destination address

Imagine multiple NEs with an attached sensor that are built into a wall and transmit data to a wireless network. At a certain moment, the protocols on the wireless network have changed and the data from the NEs in the wall can no longer be processed. It is assumed that the wireless signals (or data) can still be received by neighboring NEs, but that the destination of the data is unknown. The data is wrapped into a packet, marked with a token *unknown* and stored locally on the neighboring NEs. All this is illustrated in figure 4.2.

A programmer wants to create an application called LegacySensor that processes the packets from the sensor-equipped NEs in the wall. Now the programmer needs a way

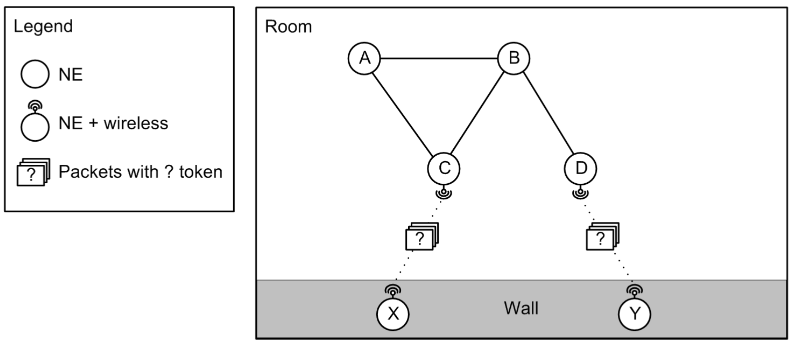

Figure 4.2: *Network Element C and D receive unknown data from old NEs that were built in the wall. The network provides a service to collect this data.*

to transfer all the packets marked with the *unknown* token to his application. Since it is a typical network service to collect all packets satisfying some condition – even the unknown packets – the DSL should be equipped with mechanisms to do so.

4.3.1 Selecting the involved Network Elements

By embedding the NC corresponding to NE A (NC^A) in the LegacySensor application, the developer has gained access to some resources in the network. For the sake of simplicity we assume that the developer already knows the NEs that contain the packets with the *unknown* token. Now the DSL needs to provide a mechanism to select NE^C and NE^D. This is implemented with the following SQL-like mechanism:

```
SELECT *
FROM NetworkElement ne
WHERE ne.identifier = "C"
OR ne.identifier = "D"
```

The above statement can be issued to any NC and after execution it will result in a collection of NEs. The *select* instruction can also be used to obtain other information from the network such as the number of connected neighbors. However it is not obvious that this result of this query will be automatically returned to NC that issued the *select* statement. Therefore, a *select* statement must always be embedded into another instruction that can process the information returned by the *select* statement. An example is the *send* instruction which will send the result to a specified destination:

```
SEND <select>
TO <destination>
```

The *destination* part can be a *select* statement that results in a collection of applications or NEs that are able to process the information that is returned by the *select* part. Figure 4.3 shows the involved concepts for performing queries on a network.

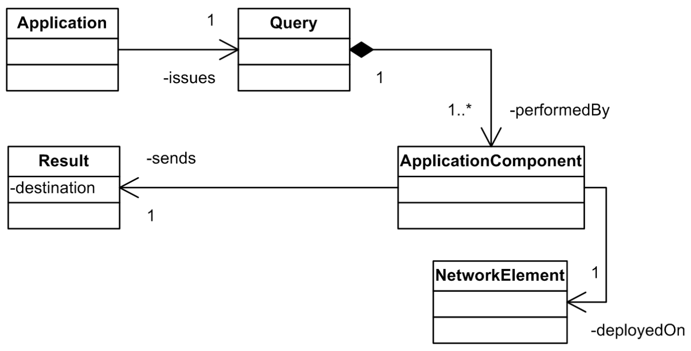

Figure 4.3: *Case 2: An application issues a query statement. This statement is compiled to an AC that queries the NE and returns the result to a destination.*

4.3.2 Creating an application-specific route

The LegacySensor application needs a way to obtain the packets with the *unknown* token from NE^C and NE^D to itself. This can be done by creating a Route from these NEs to the application:

```
CREATE ROUTE
FROM <select>
TO <select>
USING "DijkstraShortestPath"
```

The above statement creates a route from NE^C and NE^D to the LegacySensor application that exclusively allows packets marked with the token *unknown*. Since there can be multiple routes possible from NE^C and NE^D to the application, an algorithm must be specified that determines *how* the route will be created; in this case the Dijkstra shortest path algorithm is used to calculate the NEs that will be involved in the route.

When a NE contains packets with a token that match a route, the AC on it will automatically start forwarding the packets to the neighboring NE, indicated by the routing rules. So when the route is created, the packets will be transferred to the LegacySensor application. Figure 4.4 illustrates the involved concepts for creating a route.

4.4 Case 3: Network filesystem

Just like the Open Shortest Path First (OSPF) routing protocol [25] depends on routing tables, a CNS can depend on configuration files that are somewhere on the network. This case study investigates a CNS that can store files *in* the network. Following the definition of CNS in section 2.1.1, it should not be possible for an individual NE to provide the service. It is the aim to implement the storage mechanism by using existing instructions as much as possible.

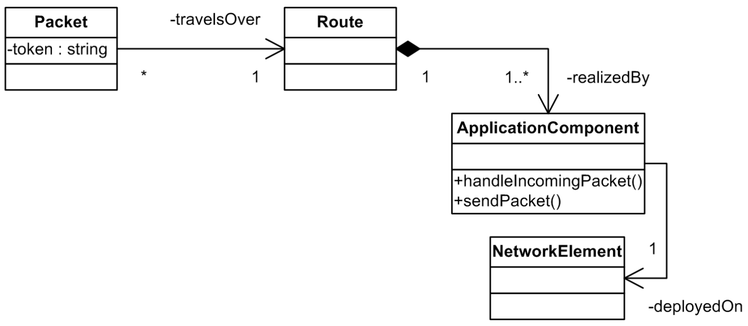

Figure 4.4: *Case 2: Packets travel over a designated route which is implemented by multiple ACs that are deployed on multiple NEs.*

4.4.1 Storing a file on multiple Network Elements

Storing a file in current networks primarily involves end-systems, such as webservers, backup systems or computers of end-users. Numerous protocols exist that can transfer files to the remote location of which the best known protocol is probably the File Transfer Protocol (FTP) [31]. Intermediate systems such as routers are usually not used for storing the files of end-users

In this case study we use the routing capabilities of NEs to store a file. To achieve this, we take a file and split it into several fragments. Each fragment is encapsulated in a packet and marked with a token to recognize it later on. The network keeps forwarding the packets from NE to NE over designated routes. Figure 4.4 from the previous case study already contains the concept of routes. Figure 4.5 shows the additional involved concepts of this case study.

So now that the file is in constant transition, traveling as separate packets with no specific destination; it is *in* the network and hence we call it an *in-network* file. Only the network as a whole can distribute and retrieve files. The following command will store a file:

```
STORE <select>
AS "my-in-network-filename"
ON <select>
PARAMETERS (param1=value1, param2=value2, ...)
```

The first *select* part indicates which file(s) will be stored in the network. The string that follows the AS clause assigns a name to the distributed file(s) that can be used as an identifier to retrieve the file in the future. The *on* clause indicates which NEs are selected to perform the forwarding of the fragments. Finally the *using* clause takes a collection of parameters that determine how the network handles the in-network file.

When a *store* command is issued, the involved NEs are configured to keep forwarding fragments of the file(s). To prevent flooding of the network, a forwarder can hold a fragment for some time and then forward it. Another option is to immediately forward the packet, but store a copy of it. When a copy of the same fragment is received again, it will be dropped. Obviously the memory that stores these fragment copies needs to be flushed regularly.

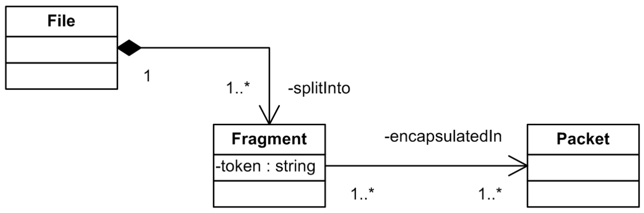

Figure 4.5: *Case 3: A file is split into multiple fragments that are encapsulated in packets. The packets are continuously forwarded over designated routes.*

4.4.2 Retrieving a stored file

There are a few ways for a NE to retrieve an in-network file. A NE can just wait and gather all the fragments of the file that travel over it. This is the most inefficient approach and will only be feasible in very small networks. Another way to retrieve all fragments is to send out an agent that 'catches' the fragments. This approach is a slight improvement to the first one, but still it is likely that not all fragments will be caught. Finally the best way is to use the mechanism from case study 4.3 that sets up routes from all involved NEs (or a sufficiently large subset) and directs all the fragments to the NE that is interested in the entire file.

Collecting the fragments of an in-network file is to the user just as obtaining any other information about the network. Case study 2 already presented the concepts of performing a query. We reuse the query concept to obtain the fragments of a file:

```
SELECT *
FROM NetworkFiles nf
WHERE nf.identifier = "my-in-network-filename"
```

4.5 Case 4: Self positioning code

This case study investigates the mobility of applications and ACs in a network. Consider the following scenario: a network management application monitors several systems in a network. When one of the systems is about to get overloaded, the application needs to take action and change the configuration of the systems. Assume the management application is similar to the one discussed in section 2.3.1 and uses the SNMP protocol [5] to monitor and control the systems.

If the network is heavily congested, the monitoring information will be delayed and the network management application may be no longer able to react in time, also because the instructions to change the configuration of the systems will be delayed as well. In such a situation, mobile applications are preferred. A mobile application can 'travel' to the area of interest and autonomously perform the monitoring and controlling of systems. Because the application is closeby, traffic between the systems and the application can be reduced to a minimum and the application is more likely to respond in time.

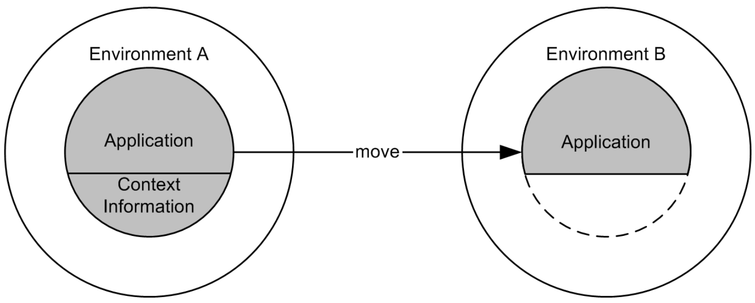

Figure 4.6: *Most applications let the environment manage their context information (program counter and memory). Unless there is support from the environment, an application can not resume execution after it was moved to another environment.*

4.5.1 Awareness of the environment

Taking and moving an application to another NE and resuming it from there, requires that its context information (memory and programcounter) is moved as well. Figure 4.6 shows an application whose runtime environment does not support resume of a suspended application, thus the context information is lost. This is the case with most applications at the moment. In such a situation, an application can only resume after moving when it manages the context information by itself, instead of the environment.

There are environments that do support the resuming of moved applications. VMware is an example of an environment that supports the moving of an entire operating system[1]. In the network service language, moving an application can be done as follows:

```
MOVE <select>
TO <select>
```

The *select* clause in the *move* statement expects a collection of applications. These will be moved and resumed on the locations that are indicated by the second *select* part in the *to* statement. If it is not possible to resume the applications, they will just be restarted. Figure 4.7 shows that the *move* statement is issued by ACs to other applications or to ACs. Again, we deviate from the standard UML class-diagram notation by using a dashed line to indicate a method invocation.

4.6 Case 5: Feedback driven recompilation

Because network topologies change, an existing CNS can break. Take for example the application described in section 4.3.2 that creates a shortest path between A and B. This application can become invalid when there appear new NEs in the network, introducing a shorter path between A and B, hereby rendering the shortest path invalid. An analogy of this situation is a PC application that is optimally compiled for the instruction set of a processor. Changing the instruction set requires recompilation so the application again fits the new hardware. A change in network topology may require recompilation.

[1]Provided that the operating system was installed on VMware

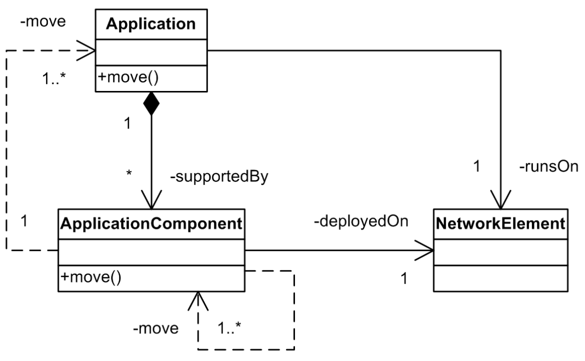

Figure 4.7: *Case 4: ACs can instruct other applications or other ACs (or themselves) to move to another Network Element.*

Several kinds of changes are possible. In this case study we only investigate topological changes.

4.6.1 Compiling for a dynamic target

In this case study, a NE creates or removes a connection and broadcasts the event to the ACs that are in the network. An AC can react to the change and ask the compiler to recompile itself or another AC. To achieve this, an AC is equiped with the source code of the CNS. The AC can issue the following statement to compile an AC:

```
COMPILE <select>
```

The *select* clause should return a collection of ACs. Every AC contains an ID that is unique to the CNS it belongs to, as well as a revision number that is incremented every compilation. Imagine a route that is realized by multiple ACs. One AC asks the compiler for recompilation. Now the compiler puts multiple new ACs on the network that have the same ID as the previous ACs, but a different revision number. A NE that receives an AC with a higher revision number will replace the old one. So the first AC that detects the change initiates the recompilation that will replace itself as well as the other ACs that contribute to the CNS. The involved concepts are illustrated in figure 4.8.

4.7 Analysis of the network service domain

The network *topology* describes the physical locations and the connections between the NEs. Sensor networks for example take advantage of the different locations of NEs by obtaining information from the region in which the NE resides. Case study 1 assumes that the topology is programmable which requires instructions to (dis)connect links between NEs.

Packets that travel from a source NE to one or more target NEs follow a path that is made up of intermediate NEs. Such a path is usually called a *route* and can be seen as a virtualization of the topology concept. A virtual topology can be created by setting up

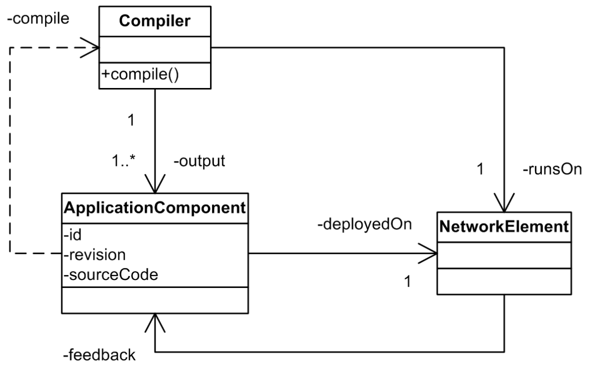

Figure 4.8: *Case 5: An AC can receive feedback from the Network Elements it runs on and as a reaction it can ask the compiler to recompile itself using its internal sourcecode.*

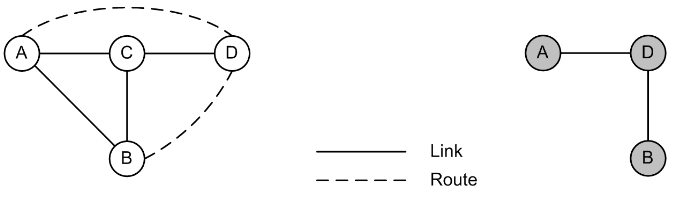

Figure 4.9: *A route can be regarded as a virtual topology. A route exists between A and D and between D and B. The gray NEs depict the virtual topology that is created by the routes.*

routes, this is illustrated by figure 4.9. Case study 2 introduces the notion of a route and because a route is the virtual representation of a physical topology, similar high-level instructions (*create*, *delete*) can be used to create or remove a route. Case study 3 uses the properties of a route to store a file *in* the network.

Since managing topologies and routes is only a small – but common – part of a CNS, managing (third party) applications is very useful when deploying network services. If the application is not known, it can only be installed, removed, started, stopped and if supported, it can be moved and resumed elsewhere. These applications can do very specific tasks that can not be programmed by a CNS language. This is illustrated by figure 4.10 which shows the overlap of the Network Domain (ND) and the Application Domain (AD).

There are many applications in the AD/ND intersection that can not be expressed with a CNS language. These applications reside in the intersection with 'Other domains'. As an example, consider an application in the audio processing domain. This application introduces a network service in which ACs start downsampling an audio file contained in a stream of packets when the network is reaching its maximum capacity[2]. In such a case, the CNS language can only be of assistance by distrubuting the ACs, or maybe to manage the topology or the routes over which the audio packets travel.

Finally we looked at the issue of compiling code to a dynamic target. This requires

[2]This example was taken from the IJkdijk project [37]

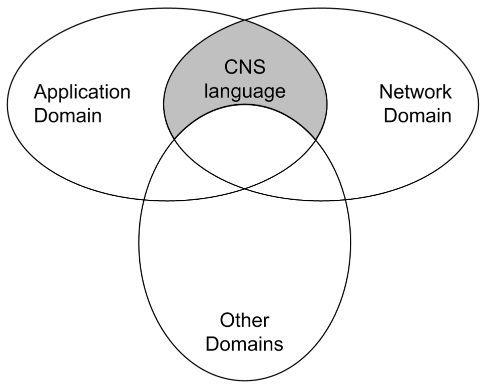

Figure 4.10: *The overlap of different domains with the networking domain. A CNS language can only be of limited support to applications in the interestion of the Application Domain with the Network Domain. For the rest, functionality is provided by applications created for 'Other domains'.*

mechanisms to detect changes as well as a way to restore applications that became invalid because of this change. Recompilation is a solution that requires few knowledge about the details of the change and can be easily implemented.

In all the case studies there is a need for a mechanism that obtains information from the network. Since routing packets is a rather obvious network service that should definitely be included in the CNS language, managing a topology should also be included since a route is just a virtual representation of a topology. For the rest, a CNS rapidly enters the 'Other domain' and a CNS language can only provide limited support by distributing and deploying the applications that contribute to this CNS.

4.7.1 A model of the Network Service domain

From the domain analysis in the previous section we conclude the the most common activities in a network are obtaining information from the network, managing topologies and routes and be of assistance to applications that are distributed in the network.

Figure 4.11 groups together the concepts of the case studies in this chapter. All the concepts in this figure are reflected in the meta-model of the language that is presented in the next chapter.

Summary of the candidate statements

In every case study there is a need for information retrieval, for this we use a *select* statement, similar to the SQL select statement. This statement can be used in combination with a *send* statement that specifies where the result of the selection should be sent. To create and remove links and routes, we use a *create* and a *delete* statement that can make use of the *select* statement. For storing information in the network, we use the *store* statement. To recompile a possibly broken CNS we include the *compile* statement.

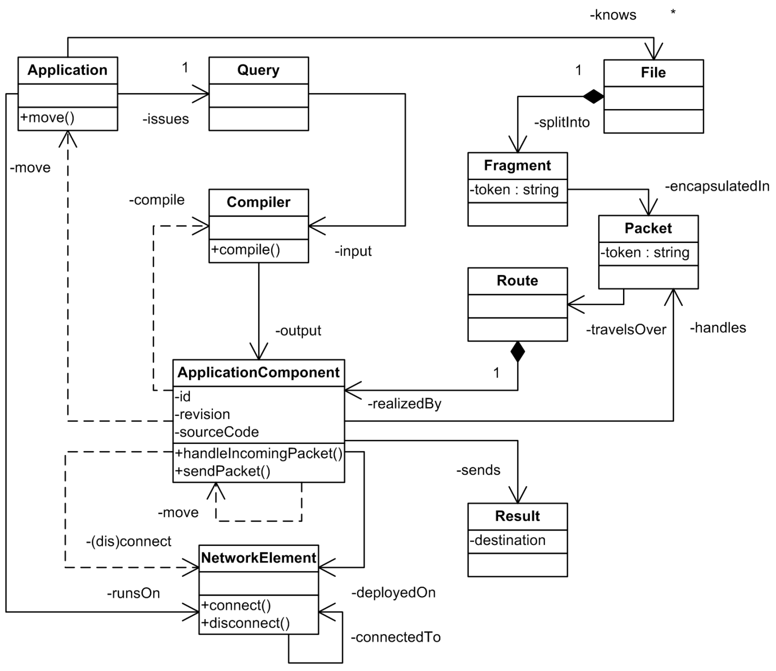

Figure 4.11: *The structure of the domain model.*

4.8 Conclusions

In this chapter we presented five case studies, each with a different demand on the network. The problem that occurs in all studies, is the retrieval of information from the network. So a statement to retrieve information is an obvious candidate. For the other problems, it is more difficult to draw the line between a CNS language and 'other domain'. By consulting with domain experts we decided to include statements for managing topologies and routes and for the rest we included statements that support third-party applications. Since these third-party applications are unknown, they are treated as black boxes that can be installed, removed, started and stopped and when supported they can be moved and resumed on another NE.

Chapter 5

CoNSoLe: A Network Service DSL

Every configuration file eventually becomes a programming language

James Gosling

This chapter presents the architecture of a programming language that can be used by applications to compose new network services. The first step in designing a language is to map important domain concepts to first class language constructs in the language meta-model. Section 5.2 presents the architecture of the language. Section 5.3 elaborates on the different areas of the architecture. Section 5.4 presents a virtual machine which is the environment in which the compiled language executes. The statements of the virtual machine are explained in section 5.5. Section 5.6 presents the compiler that translates CoNSoLe programs to statements for Application Components. The solutions provided by the language are compared with other existing solutions to similar problems in section 5.7. Section 5.8 concludes this chapter.

5.1 Introduction

This chapter and chapter 4 are the result of an iterative process and directly depend on each other. The case studies assisted in making design decisions and helped shaping the language. The developed language in turn was used to implement prototypes for the case studies. The MDE concepts from chapter 3 were used to implement the language, the virtual machine and the compiler.

A major side effect of creating a DSL is that it also serves as a *methodology* to explore and describe the research domain. This somewhat bold statement is supported by the fact that designing a domain specific language forces the developer to carefully investigate the concepts of the application domain, their relations and the environment to which the language gets translated. All these findings are explicitly recorded in models: the source language, its meta-model, the compiler and the list of statements of the virtual machine.

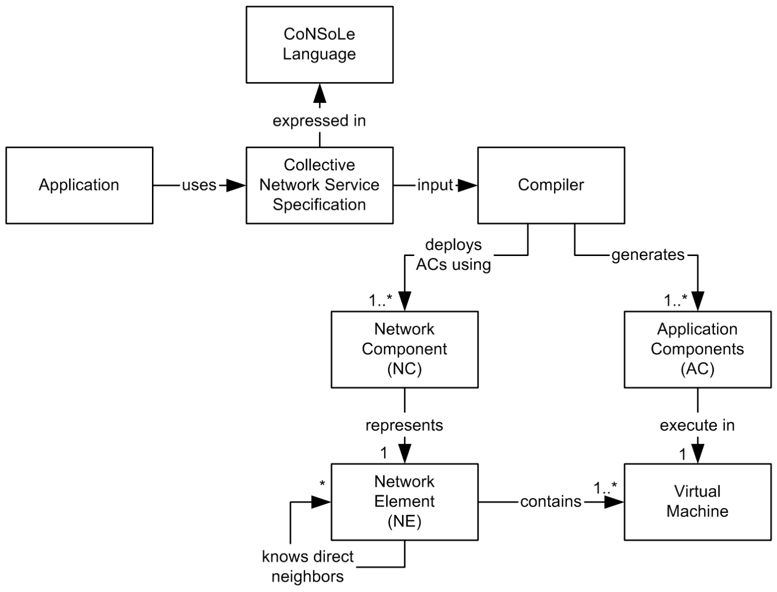

Figure 5.1: *Overall architecture of the CoNSoLe language*

5.2 The Architecture of the CoNSoLe language

A helicopter view of the architecture of the CoNSoLe language is shown in figure 5.1. An application uses the language to create or adapt a network service, for example a route between two network elements. This language serves as input to the compiler which knows the network through one or more network components (NC). From the input specification, the compiler generates one or more application components (AC) that need to be put on the network in order to actually create or adapt the network service. The compiler uses NCs to deploy the ACs to the NEs. Once the ACs are deployed, they will run in the virtual machine that is present on every NE that is prepared to work with the CoNSoLe language. In concert, the ACs will create or adapt the network service that was specified.

A network service specification is the result of the collaboration between code fragments in the CoNSoLe language and the embedding application. For example an application can use the CoNSoLe language to obtain information about the network topology before it decides whether it will create one or two routes. For the creation of these routes, it can again use CoNSoLe statements. The next section explains the structure and the meaning of the CoNSoLe language.

5.3 Collective Network Service Language

This section describes a language that reflects the essence of a CNS as defined in section 2.1.1; it must not be possible to program individual NEs using the language. This is an activity for which a GPL is a better candidate. To summarize the domain analysis

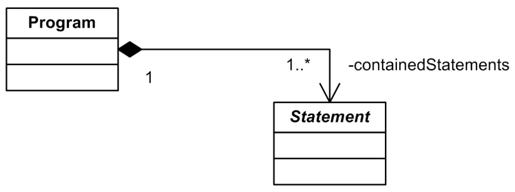

Figure 5.2: *A CoNSoLe specification consists of a collection of abstract statements.*

of section 4.7, a CNS language can be used to obtain information from the network (section 5.3.2), configure the network topology (section 5.3.3), facilitates application-specific routing (section 5.3.4) and it assists distributed applications with mechanisms to install and run the distributed components (section 5.3.5). NEs must contain facilities to contribute to the CNS. These facilities were captured in a Virtual Machine (VM) which is described in section 5.4.

5.3.1 The structure of CoNSoLe specifications

Figure 5.2 shows the structure of CoNSoLe programs. Since in most situations the language will be embedded inside another application, the syntax is simple: a program is just a collection of statements. There is no such thing as a program name, variable or constant declarations or other typical initialization constructs.

The following code fragment gives the concrete syntax of a CoNSoLe program using the TCS notation as described in section 3.6:

```
template Program main
  :  statements
  ;

template Statement abstract;
```

5.3.2 Network Information Retrieval

An application that is about to change or introduce a network service, will most probably first obtain information about the network. By doing this, the application can build a model of the network upon which it can base decisions that will cause changes to the network.

There are several ways to obtain information from the network. In a broad sense, there are two options: the first option is to let NEs periodically publish their data to a central location that can be queried (push). The second option is to directly ask (pull) all NEs. The select mechanism from the Structured Query Language (SQL)[6] that has been developed to obtain information from databases, is a DSL that is a perfect candidate for this job. SQL uses the pull approach. An alternative to SQL is to obtain information like the approach taken by the NML language described in section 2.3.1.

Which network information can be obtained depends on what information is made accessible by the NE. Figure 5.3 shows the structure of the select statement. The following code listing shows the corresponding concrete syntax:

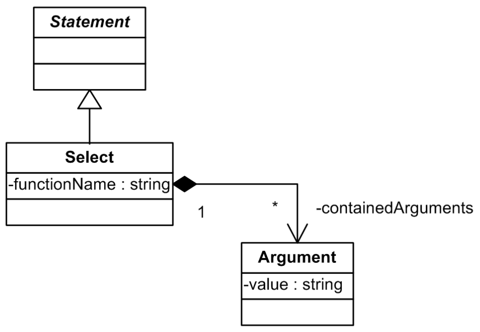

Figure 5.3: *The structure of the Select statement*

```
template Select
  :  "SELECT" functionName "(" arguments{separator = ","} ")"
  ;

template Argument
  :  value
  ;
```

The following listing shows some example usages of the select statement:

```
SELECT numberOfNetworkElements( "timeout=3min" )
```

The *select* statement is followed by a function that can take zero or more arguments. These are built-in functions that are implemented by the compiler. At the moment only the *query* function is implemented. This function delegates the actual query to an existing library that actually performs the query. For more details on how this is implemented see section 5.5.1.

Embedding the Select statement in other statements

The information that is returned by the *select* statement can be used by the application that issued the statement or it can be used by other statements that require additional information. The first can be done by using the *send* statement whose structure is shown in figure 5.4. The following listing shows the concrete syntax:

```
template Send
  :  "SEND" statement "TO" destination
  ;

template Destination
  :  ipAddress":"ipPort
  ;
```

The following listing is an example of the *select* statement in combination with the *send* statement:

```
SEND
  SELECT numberOfNetworkElements( "timeout=10sec" )
TO
  "192.168.0.101":1204
```

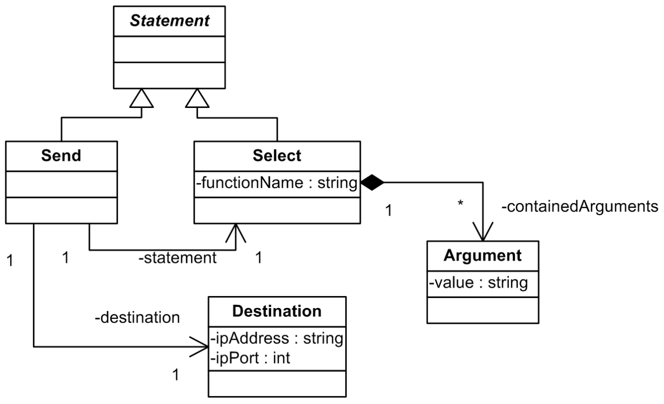

Figure 5.4: *The structure of the Send statement*

The information that is returned by the *select* statement will be passed to the *send* statement which sends the results as a collection to a specific port at a specific IP-address. The *send* can be followed by any statement that returns information. This is done because in a distributed environment like a network it is not trivial that the returned information should always be delivered back to the application that issued the statement.Alternatively, the *send* statement can also take a packet as an argument.

Since there is no default addressing mechanisms specified in the UPVN model, IP addresses were used instead to test the implementation of this statement. When a default addressing mechanism is defined, the destination part in the *to* clause can best be replaced by a *select* statement that returns a collection of network elements or a collection of applications.

5.3.3 Configuring Network Topologies

In sensor networks, topologies do matter, for example to perform localization of objects that are in the detection range of the network [11]. The initial UPVN paper [21] also contains an example in which an application finds links in a network whose disconnection will lead to a split of the network. With the ability to create additional connections, the network can be made more robust.

By using the *select* statement that is described in the previous section, collections of network elements can be rapidly indicated. With the addition of the *create* and the *delete* statement, connections can be created and removed. The structure of these statements is shown in figure 5.5. The following listing gives the concrete syntax of the *create* and the *delete* statement:

```
template Delete
  :  "DELETE" type where using
  ;

template Create
  :  "CREATE" type where using
  ;
```

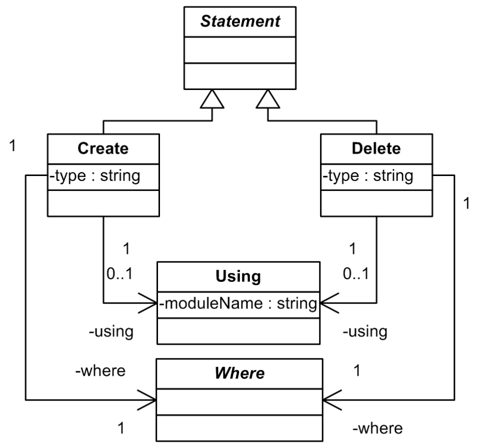

Figure 5.5: *The structure of the Create and the Delete statement. Both statements rely on the abstract Where statement.*

```

template Where abstract;
template LinkWhere
  :  ( isDefined(from) ? "FROM" from )
    "TO" to
  ;

template LocationWhere
  :  "AT" to
  ;

template Using
  :  "USING" moduleName
  ;
```

The following example will create additional links between all NEs that have only one neighbor, to the NEs that have more than 90% battery power. Replacement of the *create* statement by the *delete* statement will remove these links:

```
CREATE Link
  FROM
    SELECT query(
      SELECT * FROM NetworkElement neListA WHERE ne.neighbors.size = 1
    )
  TO
    SELECT query(
      SELECT * FROM NetworkElement neListB WHERE ne.batteryPercentage > 90
    )
```

The type argument of the *create* statement is a Link. Other types like Route are possible as well. The *create* statement is followed by an instance of the abstract *where* statement, which is either a relation between network elements (LinkWhere) or a collection of network elements (LocationWhere). This is illustrated by figure 5.6. In the above example, the LinkWhere instance is used, because the links need to be created *between* network elements.

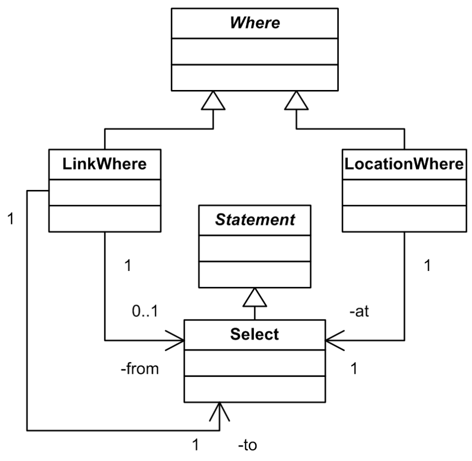

Figure 5.6: *The structure of the Where statement.*

The *from* clause uses the *query* function to pass a query to a library that can process it. Instead of sending the information returned by the *select* statement back to the application using the *send* statement, it is used as input for the *from* and the *to* statement. From the concrete syntax listing and figure 5.5, it can be seen that the *from* clause is optional. If it is left out, the links will be created from the network element that receives the CoNSoLe statement.

5.3.4 Application Specific Routing

Applications that have control over how a network routes its information, can make more efficient use of the networks' resources. For example, once an application that distributes video streams has discovered the network topology well enough, it can create an efficient multicasting tree to deliver the video content. Another example of application-specific routing is the p4p protocol [38] that is being developed to support p2p networks with application-specific routing. To spare the backbone connections in the network, clients can ask for peers that are physically nearby or in the same sub network.

The syntax to create a route is similar to the syntax to create a connection. The only difference is that we do not pass Link to the *create* statement, but Route:

```
CREATE Route
  FROM
    SELECT query(
      SELECT * FROM NetworkElement ne1 WHERE ne1.identifier = 3
    )
  TO
    SELECT query(
      SELECT * FROM NetworkElement ne2 WHERE ne2.identifier = 15
    )
  USING "DijkstraShortestPath"
```

The *using* clause is new in this example. Since it is not trivial *how* the route between the network elements should be created, a module can be referenced that will calculate the path. By doing this, the CoNSoLe language does not need graph algorithms that

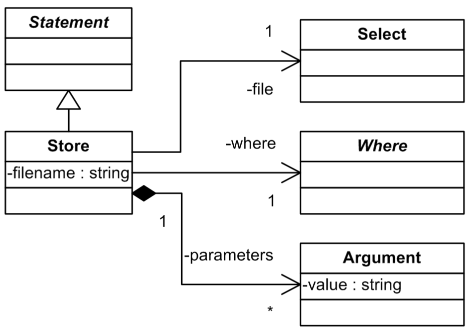

Figure 5.7: *The structure of the Store statement.*

can calculate routes between collections of network elements. In the above example the module will calculate the shortest path using the Dijkstra [9] shortest path algorithm. If the *using* clause is left out, the Dijkstra module will be used by default. For more information about modules, see section 5.4.5.

To store a file in the network, the *store* statement can be used. This statement takes a *select* clause to indicate the file that must be stored. The *where* clause indicates which NEs will be involved in storing the file. Between these NEs, special routes will be created to forward the fragments of the file. The *using* clause can be used to adjust to parameters of the *store* statement. Figure 5.7 shows the structure of the *store* statement. The following listing shows the concrete syntax:

```
template STORE
  : "STORE" select
    "AS" filename
    where "(" parameters{separator = ","} ")"
  ;
```

5.3.5 Supporting Distributed Applications

Section 4.7 underlines the fact that applications in a network (including protocol implementations) in a great deal determine its characteristics. The most basic way to deploy applications on the network is to broadcast it to every network element. This requires no knowledge of the network, which is an advantage because obtaining information costs time and the information may be incomplete. A broadcast is rather robust and will usually not fail because of the disconnection or the failure of some NEs. A drawback of broadcasting is, that it will flood the network if there are no protections against flooding.

A better way to distribute applications is to use the capabilities of the *where* statement that in turn uses the *select* statement to narrow down the number of network elements that should receive the application. Figure 5.8 shows five mechanisms to handle application deployment: *install*, *uninstall*, *start*, *stop* and *move*. The following listing shows the concrete syntax:

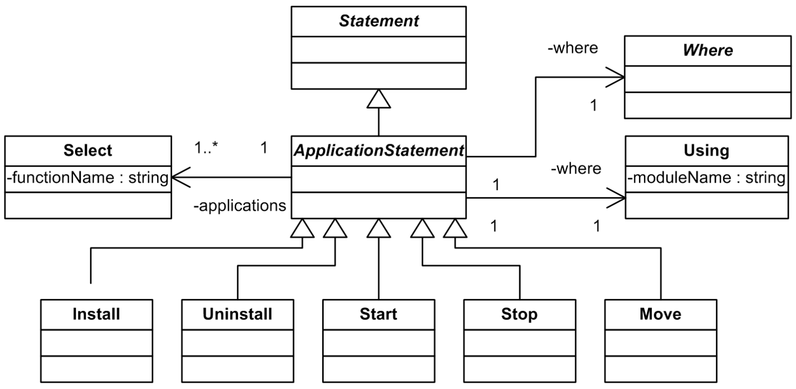

Figure 5.8: *The structure of the Install, Uninstall, Start, Stop and Move statements.*

```
template ApplicationStatement abstract;

template Install
  :  "INSTALL" select where using
  ;

template Uninstall
  :  "UNINSTALL" select where using
  ;

template Start
  :  "START" select where using
  ;

template Stop
  :  "STOP" select where using
  ;

template Move
  :  "MOVE" select where using
  ;
```

The following listing is an example that installs an application called HelloWorld onto all the NEs that have more than 300 kB of disk space using the FtpInstall module:

```
INSTALL
  SELECT query(
    SELECT * FROM Applications a WHERE a.name = "HelloWorld"
  )
  AT
    SELECT query(
      SELECT * FROM NetworkElements ne where ne.diskSpace > 300
    )
  USING "FtpInstallModule"
```

The *install* statement is followed by a *select* that must return a collection of applications. These application can reside on the NE from which the statement was issued, or on other NEs in the network. This example uses the LocationWhere to indicate the network elements on which the application should be installed.

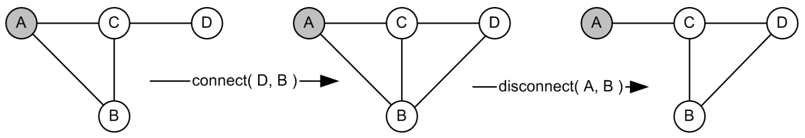

Figure 5.9: *Moving an application from A to D. Changing the entire topology is a primitive – but equal – way to move the application. From the application's perspective the same result is obtained after the operation. Assumed is that the network elements are equal and that NEs contain at most one application.*

Most applications that run on a NE can not be known in advance. As such, they must be treated as a black box. If we know a bit more about the application, for example that it can run in an environment that supports the suspending and resuming of applications, we can use the optional *using* clause to involve a module that can handle the communication with this environment. The *uninstall*, *start*, *stop* and *move* statement have the same structure as the *install* statement.

The *move* statement can only use the LinkWhere to indicate from which NEs the applications will be be moved to which other NEs. Moving an application from one NE to another is not a very common activity in current network management. However, we argue that a *move* statement is indeed a natural statement: it can be implemented by performing a series of *connect* and *disconnect* statements. Figure 5.9 shows how moving an application is essentially the same as changing the topology of the network, under the assumption that NEs are equal or contain at most one application. Since it is not very efficient to move applications this way, the move statement is implemented in an alternative way that is described in more detail in section 5.5.

For every AC, the *compile* statement issues a *compile* statement to the VM. The source code of the original CNS specification is passed as an argument.

5.3.6 Recompilation to handle dynamic topologies

From case study five (section 4.6.1) we used feedback to detect a topological change. When new NEs appear in the network, they broadcast a message with this event. A NE that loses a neighboring NE also broadcasts this event as a message contained in a packet. Every AC can use a handler to respond to this kinds of events. If necessary, an AC can issue a *compile* statement to the virtual machine which will result in the recompilation of the original CNS specification. Figure 5.10 shows the structure of this statement. The following listing shows the concrete syntax:

```
template COMPILE
  : "COMPILE" select
  ;
```

The *select* clause should return a collection of ACs. Every AC contains the original source code of the CNS. The virtual machine invokes the compiler and the compiler creates new ACs. Because the new ACs have a higher revision number than the old ACs, the old ACs will be replaced.

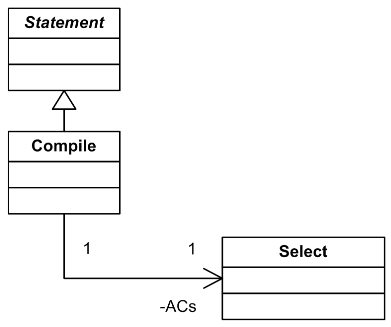

Figure 5.10: *The structure of the Compile statement.*

5.4 A Virtual Machine for Application Components

There are two major reasons to implement the runtime environment as a Virtual Machine (VM). First is the fact that the developer is forced to think about which statements to implement. Second, a VM handles the context information of applications (memory and program counter) itself. This makes implementing a feature such as code mobility a lot easier. Had we decided to generate C++ output and let an existing operating system be the runtime environment, code mobility would have been very difficult to implement.

5.4.1 The Architecture of the Virtual Machine

The architecture is shown in figure 5.11, each component of the archicture will be explained in the next sections. The Virtual Machine (VM) is implemented in Java. See appendix G for more details regarding the implementation of the VM. The main activity of the VM is executing ACs, the order of execution is determined by the *AcScheduler*. An AC contains statements that are executed by the VM and when necessary, it uses the *NetworkElement* to complete the statement.

5.4.2 The Structure of an Application Component

A compiled CoNSoLe program results in one or more Application Components (AC). Figure 5.12 shows the structure of an AC. The first part of a AC is the initialization part that declares an identifier and a revision. When a CoNSoLe program is recompiled, a new AC replaces the old one. The old AC can be found using the identifier (that remains the same over all the compilations). The revision number will be incremented. ACs with a lower revision number will be replaced by newer ones.

The next part declares the handlers. A handler assigns an event type to a function. Events can be the arrival of a packet, the disconnection of a neighbor or the notification of a clock tick. This is described in more detail in section 5.4.4. The main part of an AC is made up of functions. A function is a collection of statements. A statement can contain zero or more arguments.

The abstract syntax of ACs is specified in the KM3 language [14]. This specification can be converted to an Ecore model. Ecore models can be serialized to an XML file.

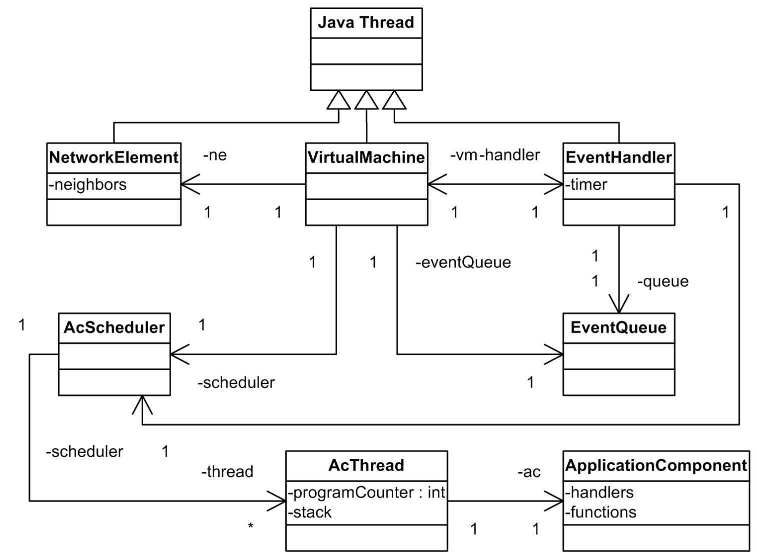

Figure 5.11: *The architecture of the Virtual Machine*

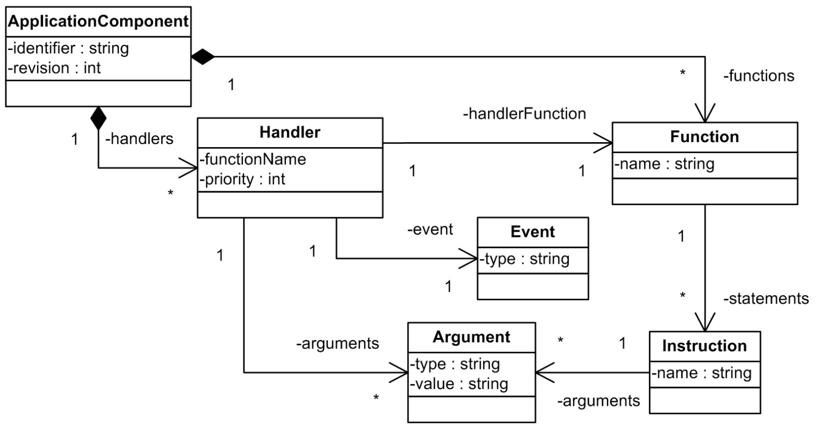

Figure 5.12: *The meta-model of an Application Component (AC).*

This XML format is used as the concrete syntax for ACs. The big advantage of this decision, is the good tool support for Ecore models by the Eclipse [3] environment. These tools allow for displaying and easy manipulation of Ecore models. Furthermore a code generator can create all the Java classes that are needed for loading, manipulating and saving Ecore models. A large part of the virtual machine is made up of this generated code.

5.4.3 Concurrency and Scheduling

One of the design decisions is whether or not to provide concurrency support for ACs. Introducing concurrency requires scheduling which opens the door for deadlock situations or conflicting operations on network traffic. Just like most runtime environments, there is no true concurrency, but it is simulated by *time-sharing*. This is done by allowing each thread a short period of execution time, so that it looks as if the threads are served simultaneously.

Not allowing concurrency puts a requirement on ACs to release resources in time. One AC that contains a blocking message handler should not prevent other ACs from handling other network traffic. To avoid complexity when designing ACs, we have decided to implement a simple scheduler that allows for the concurrent execution of ACs.

By choosing to support concurrency, we could push the idea a bit further and allow one AC to start multiple threads. But since all ACs are generated, there is no need for this. If there are multiple tasks, we just generate as much ACs as needed.

5.4.4 Event Handling

When an event occurs on the NE, it notifies the VM. The VM in turn puts the event in the *EventQueue*. This queue is continuously read by the *EventHandler* thread. For example, a packet arrives in the queue of the NE. Now a MSG_RECEIVED event will be put in the *EventQueue* by the VM. The *EventHandler* thread searches all the ACs that run in the VM for handlers that are registered to take care of MSG_RECEIVED events. The following code listing shows how a handler can be created:

```
<handlers eventType="MSG_RECEIVED"
        functionName="handleMessage"
        priority="6"/>
```

In the above code listing, the *handleMessage* function will handle the event. When there are multiple handlers – either in one AC or in multiple ACs – the AC with the highest priority handler will be scheduled for immediate execution. This program counter will be changed to the handler function. This process is repeated until all handlers have had a change to process the event.

5.4.5 Complementing the limited DSL functionality by using Modules

Since the CoNSoLe language does not implement enough statements to act as a GPL, there must be some other way to complement the limited functionality of the ACs that run inside the VM. This was done by the use of modules. A module can be programmed in Java. The functionality of a module is referred to by the *using* statement, see section 5.3.4 for an example.

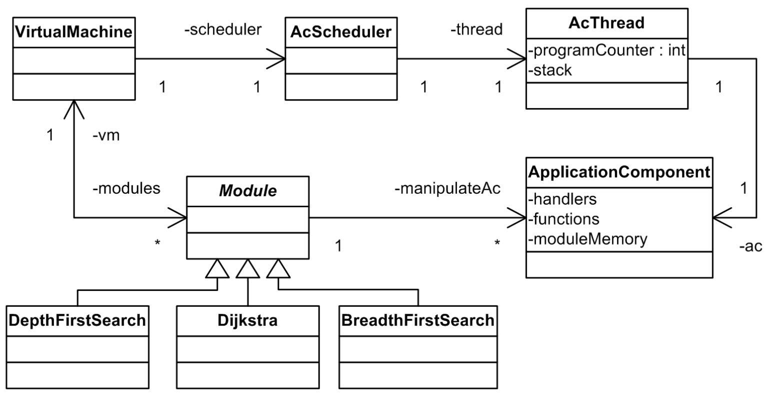

Figure 5.13: *Modules can be implemented in Java so they can complement the limited functionality of the CoNSoLe language. A module can use a dedicated memory area (moduleMemory) in the AC for its administration purposes.*

When an AC references a function from a module, it allows the module to manipulate its contents and its memory. This is illustrated by figure 5.13 which is a refinement to the simplified architecture presented in figure 5.11.

5.4.6 Control flow

A program with no loops is guaranteed to terminate. Suppose we have a compiler somewhere in the network and an AC somewhere else. We could create an AC that will be activated by an event, and just before termination it requests for recompilation. Now there is no need for a loop, because everytime something happens, the program will just be recompiled and ready for the next event.

However, when an AC must be able to respond in time, recompilation may take too long. This requires local autonomy of the AC, so that it is able to manage its own local problems and does not depend on the recompilation arranged by a compiler somewhere else in the network. For this reason, we conclude that a loop is essential for locally autonomous ACs.

5.5 The statements of the Virtual Machine

This section presents all the statements that are implemented by the VM. The statements are presented in the same order as the categories of the CoNSoLe language: information retrieval, topologies, routing and distributed applications.

5.5.1 Network Information Retrieval

Implementing a complete query language takes much time. Because of this, we decided to implement this using an existing library named: Hibernate Query Language (HQL)[1]. HQL can perform queries over the contents of any collection of objects. This is done by mapping these objects to a relational database and transforming the HQL query into an SQL query. Since all experiments were applied to a virtual network, every network was represented by an object, and therefore every detail about the network can be queried. Table 5.1 shows the only statement that was needed to perform a query.

Instruction	**Parameters**	**Return value**	**Description**
select	query	Set<value>	Rich instruction that can obtain all accessible data from the NE and sends the result to a set of NEs

Table 5.1: *Collective Network Service - Network information retrieval instructions*

5.5.2 Configuring Network Topologies

Table 5.2 shows that there are only three statements necessary to configure a network topology. All three statements are delegated to functions of the network element. The *connect* and *disconnect* statements only work in wireless networks whose NEs support the creation and deletion of wireless links. The *get_neighbors* statement can be used both on wired and wireless networks.

Instruction	**Parameters**	**Return value**	**Description**
connect	Set<NE>	-	Establish links to all NEs in the given set
disconnect	Set<NE>	-	Disconnect the links to all NEs in the given set.
get_neighbors	-	Set<NE>	Obtains a set of neighbors that have a link to the NE that executes this instruction

Table 5.2: *Collective Network Service - Topology instructions*

5.5.3 Application Specific Routing

Table 5.3 shows the statements that are directly related to packet processing. The *send* statement takes a packet and a collection of NEs to which the packet will be send. If the given collection is empty, the packet will be broadcasted to all neighbor NEs. The packet can contain additional information that will impact the behavior of the *send* statement.

Instruction	**Parameters**	**Return value**	**Description**
send	packet, Set<NE>	-	Send a packet to a set of NEs
store	packet	-	Store a packet that in the packet memory
contains	packet	boolean	Check if the packet memory already contains a copy of the given packet
reset_mem	-	-	Clear the memory that was filled by *store* instructions.
route	Set<NE>, module	-	Create a part of a route to every neighboring NE in the set, using a module that calculates the route

Table 5.3: *Collective Network Service - Routing instructions*

The *store* and *contains* statements are directly from the case-study in section 4.4. We believe these statements belong to a basic CNS statement set because they allow to implement a basic, but controlled way of broadcasting. With these statements a router can remember and check for recently received packets. This can prevent a broadcast from flooding the network, because once a router receives a packet it already broadcasted before, it can safely drop the packet.

The *route* statement creates a route between two collections of NEs. In the most basic case, both collections contain only one NE. If there are more NEs in the second collection, routes will be created from one NE to all the others, forming a tree. If both collections contain more than one NE, the first NE from the first collection is taken and routes will be created to all the NEs in the second collection. Now the second NE from the first collection is taken and again routes will be created to all the NEs in the second collection etc. The module argument largely determines *how* the route will be created.

5.5.4 Supporting Distributed Applications

Table 5.4 shows how a NE can provide support for hosting applications. Since most applications can not be moved while they are running, the *move* statement will just *stop* the application, *move* it to another NE, where it can be started with the *start* statement. There is an exception for CoNSoLe applications, because we know they can be moved while they are running. In this case the *move* statement will invoke the *serialize* statement, *move* the frozen application to other NEs where it will be *deserialize*'d and started.

Instruction	**Parameters**	**Return value**	**Description**
install	Application/AC, Module	-	Installs the Application/AC on this NE using the given module.
uninstall	Application/AC, Module	-	Uninstalls the Application/AC from this NE using the given module.
start	Application/AC, Module	-	Starts the Application/AC on this NE using the given module.
stop	Application/AC, Module	-	Stops the Application/AC on this NE using the given module.
move	Application/AC, Set<NE>, Module	-	Moves the Application/AC from this NE to the NEs in the given set using the given module.
serialize	AC	-	Serializes an (running) AC to a textual representation.
deserialize	AC	-	Deserializes an AC from a textual representation.
compile	AC	Set<AC>	Compiles the sourcecode contained in the given AC to a new collection of ACs

Table 5.4: *Collective Network Service - Application and AC related instructions*

Most statements can use a module. For example a module for the VMware runtime environment can implement the install, start, stop, suspend or resume statements. Now an application that runs inside VMWare can issue a CoNSoLe statement that will move the operating system in which it runs to another VMWare application on another NE.

5.5.5 Basic statements

With only the statements of the previous sections, an AC would be no much more than a configuration file. Table 5.5 presents some basic statements that allow for the creation of autonomous applications by providing control-flow and some simple arithmetic functions. The statements can be found in almost any virtual machine and are therefore only explained in the table.

Instruction	**Parameters**	**Return value**	**Description**
create_packet	-	-	Creates a packet from the contents on the stack
push	Value or Variable	-	Pushes the given value on the stack. If the argument is a variable name, the value of that variable will be pushed on the stack
pop	Variablename (optional)	-	Pops the top value from the stack. If a variablename argument is given, the value will be assigned to this name.
print	-	-	Pops the top value of the stack and prints the contens (For debugging purposes)
add		Result	Pops and adds the two top values of the stack. The result of the addition will be put back on the stack.
eq	-	Result	Pops and compares the two top values of the stack. If they are equal, 1 will be put on the stack, otherwise 0.
label	Label name	-	An instruction that marks a location in the code. Allows for jumps in the code.
if_true_goto	Label	-	Pops the top value from the stack, if it is non-zero performs a jump to the given label.
fragment	File	Packets	Pushes the file from the strack and fragments it into packets which are pushed back on the stack.

Table 5.5: *Basic instructions*

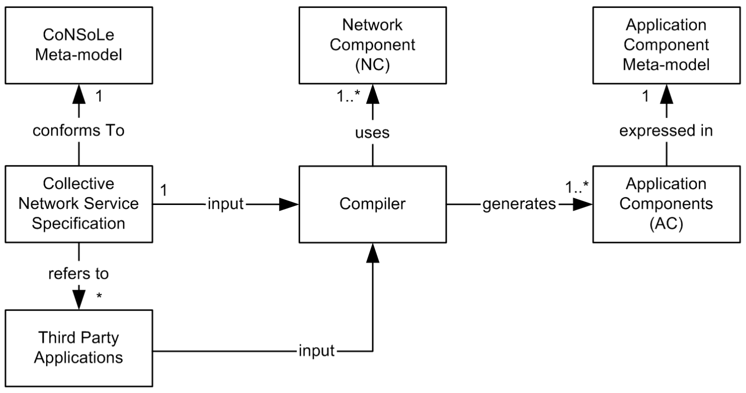

Figure 5.14: *Overall architecture of the compiler.*

5.6 Compiler

5.6.1 The Architecture of the Compiler

Figure 5.14 shows the architecture of the compiler. The compiler takes as input a program expressed in the CoNSoLe language. This language can refer to external applications, which is explained in section 5.3.5 and in section 5.5.4. The CoNSoLe program will be compiled to one or more ACs. The AC meta-model is also depicted in figure 5.12. The compiler uses a NC to obtain information about the network as well as to deploy ACs and applications to the individual NEs.

5.6.2 Compiling CoNSoLe statements

Table 5.6 shows an overview of how CoNSoLE statements are translated to AC statements.

select The *select* statement is compiled to a HQL query. In our prototype, all the virtual NEs are Java objects that run in the same Java Virtual Machine. Therefore every NE can answer queries about the network. This is explained in more detail in appendix G.

send The *send* statement is compiled to a set of statements outlined in table 5.6, that create a packet that contains the return values of another statement. First the statement argument is evaluated. This statement may leave values on the stack. These values will be encapsulated in a User Datagram Protocol (UDP) packet and send it to some port on an IP-address.

create The *create* statement compiles to a series of *connect* instructions, for every involved NE, except for the last NE. This is similar with routes, but instead of compiling to a *connect* instruction, an AC is deployed that forward packets to the next NE in the path.

When the *create* statement is not issued to the NE that maintains the connection, or is the beginning of the route, additional work has to be done. This is described in section 5.6.4. Creating a route requires a model of the network, otherwise it is impossible to calculate the shortest path between the source and target NEs. Discovering this network model is explained in section 5.6.3.

store The *store* statement compiles to the *fragment* statement and a series of *send* statements. The first statement fragments the file(s) into packets. Then it creates a number of routes dedicated to transport these packets. It relies on the *create* statement to create the routes. Then it uses the *send* instruction to put the packets on these routes.

install, uninstall, start, stop The *install* statement is similar to the *uninstall*, *start* and *stop* statement and is compiled to equally named statements – *install* in this case – for every involved NE. Then the *select* clause is evaluated to a collection of applications that must be installed. Then the second *select* clause is evaluated which is part of the *where* statement. If the AC would be an agent, then for every application, the code in the second column would created and executed at every NE of the *where* clause.

move The *move* statement is compiled to a series of statements that *stop* and *uninstall* the application and then *install* and *start* the application on another NE. Moving running code is only possible with ACs. This is shown in Appendix E.

5.6.3 Central and Decentral Network discovery

Given a network service specification, the compiler calculates the behavior (or statements) for the individual NEs. This requires knowledge about the network. There are two ways to discover the network: central and decentral. A central approach puts a demand on the compiler to (partially) discover the network. This can be done by performing a depth-first or a breadth-first search on a NC. The only requirement is that a NC can return the NCs of its neighbors. A decentral approach frees the compiler from having to obtain a (partial) model of the network. This can be done by generating ACs that have to ability to autonomously discover the network. Such an application is usually called an *agent*. The algoritm for autonomous discovery is contained in the modules as described in section 5.4.5. We use decentral network discovery in the implementations of the case studies. The next section explains which method is preferred in which situation.

5.6.4 Possible compiler strategies

Depending on the circumstances of the network, the compiler can decide to generate different output, given the same input. This relates to the initialization part of the AC that finds the right NE before it starts executing. Imagine an application where information about Network Element A (NE^A) needs to be communicated to an application on NE^D. Figure 5.15 shows three strategies how the compiler can address this problem. The first strategy is to put an application on NE^A and on NE^D that communicate with each other. The second strategy shows an agent on NE^A that travels to NE^D. The

CoNSoLe statement	**Application Component Instructions**

SELECT
query(hqlQuery)
	PUSH (hqlQuery)
	QUERY

SEND statement TO ip:port	PUSH port PUSH ipAddress CREATE_PACKET PUSH udp SEND
CREATE Link where using	PUSH where CONNECT using
CREATE Route where using	PUSH where ROUTE using
STORE selectResult	FRAGMENT file CREATE route SEND
DELETE Link where using	PUSH where DISCONNECT using
INSTALL select where using	PUSH selectResult INSTALL using
MOVE select where using	PUSH application STOP PUSH application UNINSTALL PUSH application INSTALL PUSH application START
COMPILE select	COMPILE selectResult

Table 5.6: *An overview of how CoNSoLE statements are translated to AC statements.*

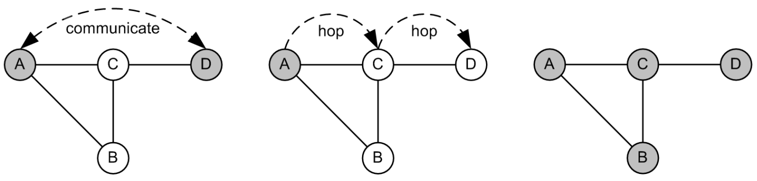

Figure 5.15: *The compiler can use three strategies – or a hybrid variant thereof – to distribute ACs in the network. The first shows two agents that communicate with each other. The second shows an agent that travels to the NE of interest. The third shows a situation in which every NE contains an agent.*

third strategy shows a brute-force solution where every NE contains the application. Table 5.7 shows the main advantages and disadvantages of each strategy. This list is not complete but only contains the main concerns.

```
DELETE Link
  FROM
    SELECT query(
      SELECT * FROM NetworkElement ne1 WHERE ne1.identifier = A
    )
  TO
    SELECT query(
      SELECT * FROM NetworkElement ne2 WHERE ne2.identifier = C
    )
```

Consider the network topology of figure 5.15. Now imagine that we issue the above CoNSoLe program to NC^A. This code would simply compile to “PUSH C; DISCONNECT”. However, additional effort is required if the same program was issued to NC^D, since it can not remove the link between A and C. To solve this issue, the compiler can apply one of the three strategies depicted in figure 5.15. Appendix D describes a solution for case study 4.2 by putting additional code in the AC that causes it to explore the network until it finds the NE on which it can perform the disconnection of the link.

Strategy	**Advantage**	**Disadvantage**
1. Communicate	Reduced bandwidth. E.g. an statement to disconnect a link costs less bandwidth than uploading an AC that will disconnect the link.	Can get isolated when links disappear
2. Agent	Can perform tasks autonomously. E.g. network discovery	Traveling takes time. Not efficient when agents grow large.
3. Everywhere	Quick response, fail safe because of redundancy.	Wasted storage space because of the redundancy.

Table 5.7: *The advantages and disadvantages of the three compiler strategies.*

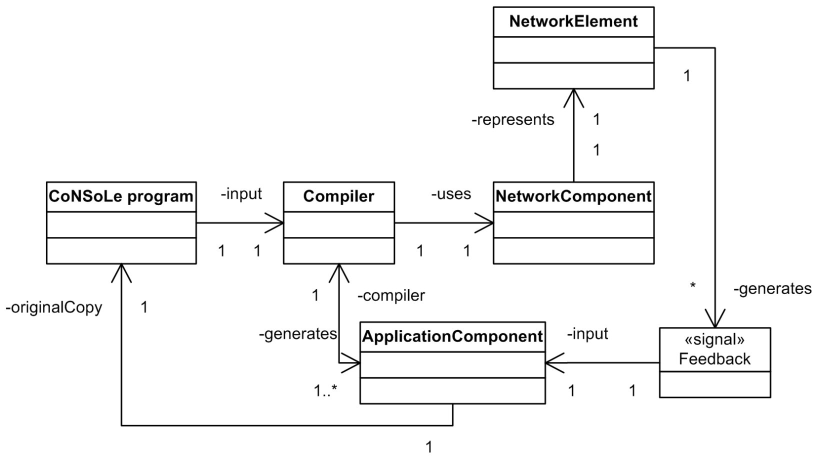

Figure 5.16: *An AC maintains a copy of its own source code. When it detects a change in the network it instructs the compiler to recompile itself.*

5.6.5 Dealing with the dynamics of the network

Case study 4.6.1 illustrates an application that creates a shortest path (route) between two NEs which becomes invalid when a new NE appears in the network. To deal with this issue, we had to extend the architecture of the compiler. Figure 5.16 shows how this problem can be addressed. The NE broadcasts feedback signals that can be detected by ACs. ACs that can respond to feedback contain a copy of their own source code. In the case of a change in the network an AC can instruct the compiler to recompile itself.

5.6.6 Implementing the Compiler as a Model Transformation

The compiler has been implemented using the Atlas Transformation Language (ATL), described in section 3.4. The model transformation takes as input the meta-model of the CoNSoLe language as well as the meta-model for an AC. Then it transforms the elements of a given CoNSoLe program into one or more ACs. When the network is discovered *centrally*, the compiler also needs a model of the network as input. In our implementation we implemented a *decentral* approach and generated agent ACs. As stated before this frees the compiler from having to know the network. For a broader view of the implementation, see appendix G.

5.7 Comparison with existing work

In this section we will compare the solution: the CoNSoLe language, with existing work that solves similar problems. We will compare the following areas of the CoNSoLe language with existing work: obtaining network information, manage topologies and routes, supporting distributed applications and using recompilation to recover from errors in the CNS specification.

Obtaining information from the network is the most important aspect of the solution. TinyDB (section 2.3.3) is a DSL entirely created to obtain information from the network. If CoNSoLe would be implemented in a real network, TinyDB would be a good candidate to replace the HQL library that is currently used to obtain information. All the other languages in chapter 2 also contain mechanisms to obtain network information. In our prototype, we can not use triggers and timing intervals like in TinyDB.

Changing the topology of a wireless network is no new activity. For example, when necessary the routing protocols in TinyDB networks can automatically establish connections to NEs that are nearby. These routing protocols try to establish a tree structure in which messages are always send to the root of the tree. There are also other protocols that manage network topologies, for example in this article by Frey [10]. We are not aware of any languages that allow the end-users to change the wireless network topology for themselves, however this will not be a technical problem. On the other hand, the CoNSoLe language can create tree routing structures like TinyDB, unless a module is created that takes care of this task.

Creating application specific routes can be done with Active Networks [35] (section 2.2.2). The route can be determined by the executable code that is contained in a packet or by the switchlets that are installed on the routers. Our approach is different with regard to the executable code. We use a global statement from which the compiler calculates the NEs that should receive their – possible customized for that NE – code. In active networks, this code must be specified in detail by the user, also the distribution of this code must be programmed. Configuration routes is in general a manual activity, also because it is an activity primarily for network maintainers. Our approach would be a step towards creating routes in an algoritmic fashion.

Distributing and deploying third-party applications is currently hard to do, since it requires a generic runtime environment on every NE. A possible generic environment could be VMWare [40], which already supports installing, starting, stopping and moving of operating systems. These commands can be issued by an external application to the VMWare Application Programming Interface (API), hereby coordinating the location of operating systems in the network. Where the VMWare API only contains the low-level instructions to move operating systems, the CoNSoLe language contains the high-level instructions to obtain information as well as instructions to move collections of operating systems, given a set of constraints.

In this research we use recompilation to recover from an invalid CNS, caused by the modification of the network topology. A similar solution is found in [39], where faults detected in networked Field Programmable Gate Arrays (FPGA) are automatically restored by reconfiguring the FPGAs. Using recompilation is a simple and interesting strategy since it does not require detailed knowledge about the kind of error or complicated recovery procedures. In our approach, the change detection is different. In [39], recompilation takes places after a fault has occured. We can react a bit earlier because of the feedback signals that are broadcasted before possible faults can occur.

In general we can say that the individual aspects of the solution are not new, however putting these solutions together in one network language has not been done before to the best of our knowledge.

5.8 Conclusions

In this chapter we presented a language for adapting and creating network services. The *select* statement turned out to be the most used statement in the language. We implemented this statement, so that every NE could respond to the query. This is a major advantage since it abstracts from the problem of distributing a query in the network and gathering the results.

We presented a VM in which ACs can be executed. The limited expressiveness of the DSL is compensated by introducing modules that can be programmed in a GPL. The statement set of the VM is described and besides the basic statements, it reflects the essentials of most network services.

The main activity of the compiler is translating statements that were issued to multiple NEs to the statements for each individual NE. For the rest the translation is rather straightforward. By the introduction of feedback from the network we made an initial approach for dealing with the problems of compiling to a dynamic target.

Chapter 6

Conclusion

Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience.

Roger Bacon

In this chapter we will have a meta-discussion regarding this work in section 6.1. In section 6.2 the answers to the research questions will be presented.

6.1 Introduction

In this thesis we have approached network programming from an entirely different perspective. Where normal network development starts working from individual network elements, we approach the network and its services as a whole. The CoNSoLe language has many similarities with SQL. We think the reason is, because both languages are developed to retrieve or change the collective properties of a set; e.g. updating the salaries of all employees in a database is similar to instructing every NE to install some application.

We have also seen that the *select* statement as the basic operation for obtaining information, is of much importance for other statements as well as for the end-application. This statement can be constrained, which allows to create a (small) model representing a (small) part of the network. Another statement or an end-application can make subsequent decisions based upon this model.

When we look back at the case studies, CoNSoLe can are already perform many collective actions to the network with only a few instructions. This list of instructions may be expanded to improve or extend the language in certain areas, possible to perform other network related tasks. But when it comes to information retrieval, topologies, routes and distributed applications we have the set of necessary instructions.

Judging by the cooperation between Cisco and VMWare to improve the migration of virtual machines [7], the part of the language that manages third-party applications on NEs is indeed relevant. It can be expected that new applications will arise that automatically coordinate the deployment of operating systems that run in environments like VMWare. There can be several reasons for moving operating systems, e.g. disaster recovery or lower electricity costs in another virtual datacenter.

We recognize that there are network related problems in which a network DSL can not provide any assistance. To overcome a part of this problem, we introduced modules that allow external developers to implement their own ideas. Also, a module is not a *collective* network service, but rather a local system library that contains often required functionality.

6.2 Answering the research questions

In this section we give an overview of the research questions. First we will give a short answer in quotes ("...") followed by a more detailed explanation. The first four research questions will be related to the domain of network engineering. The final two research questions are related to the domain of DSL engineering.

Network engineering related research questions

What is programming a network service? "Creating and distributing applications that make use of the capabilities of a NE and in concert produce a service for the end-user."

Changing a network's behavior requires a programmable network, for this reason IP-based networks are not suitable since their behavior is predetermined. Alternative programmable network models are for example Active Networks and UPVN. We picked the latter because it allows us to determine our own runtime environment for our applications.

What is the architecture of a network service language? "There are four major activities when dealing with network services: topology configuration, application specific routing, supporting distributed applications and network information retrieval"

The primary interest of this research are the primitive instructions that are required for composing a CNS. The first decision was to choose between a GPL and a DSL. By developing a DSL instead of a GPL we avoided inconsciously putting network service knowledge into a GPL, whereas with a network service DSL, *every* detail related to a network service explicitly ends up in the language, the compiler or the runtime environment. Therefore we can make a clear analysis of the instructions that are common for programming network services. More details regarding the architecture can be found in chapter 5.

What are the essential/common instructions to compose network services? "See section 5.5 for an overview of the instructions that are relates to obtaining information, manipulating topologies and routes and to support distributed applications."

The main activity of a CoNSoLe program turns out to be providing information about the network to the end-user, so the end-user is able to create a model of the network upon which it can base subsequent decisions. Obtaining information requires constraints, because in the extreme case of an infinitely large network, the model of the network will grow infinitely large. For this reason the *select* instruction that acts similar to the SQL select statement was included in the language because it can constrain the amount of desired information.

If we regard a network simply as a collection of connected NEs, the concept of a topology arises, as well as the virtual representation of a topology: routes. Besides obtaining information a CNS language should be able to manipulate topologies and routes. All the other behavior of a network is determined by applications that are too specific and reside in an other domain, to which a CNS language can not be of any assistance, except for deploying the application to the required NEs.

Is there an instruction set that can support all network services? "This answer relates to the instruction set of the CoNSoLe language: yes, if we consider topologies and routes and packets. No, if we include operating on the contents of packets."

This question is concerned with the completeness of the discovered instruction set. If we consider topologies, routes and applications as the entire domain of network services, the CoNSoLe language contains all the essential instructions. If we widen this domain, additional instructions are required. Widening the domain also involves the risk of gradually turning the DSL into a GPL.

DSL engineering related research questions

How to cope with the dynamics of the network? "Applications do not need knowledge about the network topology. When the topology changes the compiler recompiles the application for the new topology. The change-detection can be done by the application itself or by the compiler. (This is possible because the compiler is domain specific, e.g. the Java compiler will never be able to do this)"

This is an interesting contribution to the field of DSL development. Because of the overlap with the network domain, the compiler faces the challenge of compiling to a dynamic target. We only addressed the issue of a dynamic topology by detecting the changes and recompiling the applications. Recompilation allows the compiler to restore possible topology related defects that were caused by the changes in the network.

How can DSL design help understand the concepts of the application domain? "The virtual machine will reflect all essential service elements"

The sixth sub question is concerned with DSL development as a method for doing domain analysis. Developing a DSL from scratch requires a careful investigation of the problem domain. Every statement that is included in the DSL, reflects a piece of knowledge about the domain. Note that the language is best implemented as a prototype, because creating a fully functional and stable DSL requires too much time when the DSL development process is only used for performing a domain analysis.

6.3 Future work

6.3.1 ATL model transformations

The compiler was the most difficult part to implement. ATL model transformations assume that all knowledge is contained in the input models and that the transformation rules and the Object Constraint Language (OCL) are sufficient to perform the model transformation. For example, ATL can not generate a timestamp that can be used as an identifier. There are model transformations that support third party libraries for these purposes, but ATL does not support this. A workaround is to shift this problem to the input model(s). For example, an element that contains the random identifier can be added to the input model. Better cooperation between ATL and other programming languages would be a big improvement.

Another problem is the refactoring of models. At the moment, renaming an element in a meta-model renders all its instances invalid. This and similar change-related problems can be improved by better tool support.

6.3.2 Compiling to a dynamic target

In this research we only focused on the dynamics of the network topology. There are more dynamic aspects to a network that may cause errors in a CNS. Similar issues occur in the field of self-healing operating systems [34] where the operating system automatically detects and tries to recover from errors. Further research is needed in which situations the solution of recompilation is feasible.

Bibliography

[1] C. Bauer and G. King. *Java Persistence with Hibernate.* Manning Publications Co., Greenwich, CT, USA, 2006.

[2] J. Bézivin, E. Breton, P. Valduriez, and G. Dupé. The atl transformation-based model management framework. Research Report 03.08, IRIN, University of Nantes, 2003.

[3] F. Bott, editor. *ECLIPSE an integrated project support environment.* Peter Peregrinus, Hitchin, Herts., UK, UK, 1989.

[4] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework.* Pearson Education, 2003.

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (snmp).

[6] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA, 1974. ACM.

[7] Cisco Systems, Inc. Network Implications of Server Virtualization in the DataCenter, 2007.

[8] A. v. Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[9] E. W. Dijkstra. *The problem of the shortest subspanning tree.*, chapter 22. Prentice-Hall, 1976.

[10] H. Frey. Scalable geographic routing algorithms for wireless ad hoc networks. *Network, IEEE*, 18(4):18–22, 2004.

[11] S. Gangadharpalli, U. Golwelkar, and S. Varadarajan. A topology based localization in ad hoc mobile sensor networks. In R. Battiti, M. Conti, and R. L. Cigno, editors, *Wireless On-Demand Network Systems*, volume 2928 of *Lecture Notes in Computer Science*, pages 16–28. Springer, 2004.

[12] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Network programming using PLAN. *Lecture Notes in Computer Science*, 1686:127–??, 1999.

[13] R. I. G. Hughes. The ising model, computer simulation, and universal physics, 1999.

[14] F. Jouault and J. Bézivin. Km3: a dsl for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, pages 171–185, Bologna, Italy, 2006.

[15] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.

[16] S. Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, unknown 2002.

[17] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[18] I. Kurtev, J. Bézivin, F. Jouault, and P. Vulduriez. Model-based DSL Frameworks, Oct. 2006.

[19] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[20] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, June 2003.

[21] R. J. Meijer, R. J. Strijkers, L. Gommans, and C. de Laat. User Programmable Virtualized Networks, 2006.

[22] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages, 2005.

[23] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[24] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[25] J. Moy. The OSPF specification, 1989.

[26] R. Müller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 41(3):145–158, 2007.

[27] T. Parr and R. Quong. ANTLR: A predicatedLL (k) parser generator, 1995.

[28] H. Paulino and L. Lopes. A service-oriented language for programming mobile agents. In P. Stone and G. Weiss, editors, *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1294–1296. ACM Press, 05 2006.

[29] A. Perlis. Epigrams on Programming. *SIGPLAN Notices*, 17(9):7–13, September 1982.

[30] D. M. Piscitello and A. L. Chapin. *Open Systems Networking: TCP/IP and OSI.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[31] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), Oct. 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[32] J. Schonwalder and H. Langendorfer. Tcl extensions for network management applications, 1995.

[33] E. Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003.

[34] M. W. Shapiro. Self-healing in modern operating systems. *Queue*, 2(9):66–75, 2005.

[35] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), 1996.

[36] U. Warrier and P. Relan and O. Berry and J. Bannister. A network management language for osi networks. *SIGCOMM Comput. Commun. Rev.*, 18(4):98–105, 1988.

[37] Wikipedia. IJkdijk — Wikipedia, The Free Encyclopedia, 2008. [Online; accessed 22-April-2008].

[38] H. Xie, A. Krishnamurthy, A. Silberschatz, and Y. R. Yang. P4P: Explicit Communications for Cooperative Control Between P2P and Network Providers , 2008.

[39] W. Xu, R. Ramanarayanan, and R. Tessier. Adaptive fault recovery for networked reconfigurable systems. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 143, Washington, DC, USA, 2003. IEEE Computer Society.

[40] D. Zimmer. *VMware Server and VMware Player. The way forward for Virtualization.* BoD, 2006.

Appendix A

CoNSoLe Textual Concrete Syntax

This appendix presents the syntax of the CoNSoLe language. The Textual Concrete Syntax notation is used to describe the syntax [15].

```
syntax nql {

        primitiveTemplate identifier for String default using NAME:
                value = "%token%";

        primitiveTemplate stringSymbol for String using STRING:
                value = "%token%",
                serializer="’\’’␣+␣%value%.toCString()␣+␣’\’’";

        primitiveTemplate integerSymbol for Integer default using INT:
                value = "Integer.valueOf(%token%)";

        primitiveTemplate floatSymbol for Double default using FLOAT:
                value = "Double.valueOf(%token%)";

-- BEGIN Class templates

        template Program main
                : statements
                ;

        template Statement abstract;

        template Select
                : "SELECT" functionName "(" arguments{separator = ","} ")"
                ;

        template Argument
                : value
                ;

        template Send
                : "SEND" statement "TO" destination
                ;

        template Destination
```

```
                : ipAddress":"ipPort
                ;

        template Delete
                : "DELETE" type where using
                ;

        template Create
                : "CREATE" type where using
                ;

        template STORE
                : "STORE" select
                  "AS" filename
                  where "(" parameters{separator = ","} ")"
                ;

        template Where abstract;

        template LinkWhere
                : ( isDefined(from) ? "FROM" from )
                  "TO" to
                ;

        template LocationWhere
                : "AT" to
                ;

        template Using
                : "USING" moduleName
                ;

        template ApplicationStatement abstract;

        template Install
                : "INSTALL" select where using
                ;

        template Uninstall
                : "UNINSTALL" select where using
                ;

        template Start
                : "START" select where using
                ;

        template Stop
                : "STOP" select where using
                ;

        template Move
                : "MOVE" select where using
                ;

        template COMPILE
                : "COMPILE" select
                ;

-- END Class templates
```

Appendix B

Solution to Case Study 1

This case study was implemented using the *Agent* strategy, described in section 5.6.4. The following code was issued to the compiled and tries to disconnect NE 5 from NE 6:

```
DELETE Link
  FROM
    SELECT query(
      SELECT * FROM NetworkElement ne1 WHERE ne1.identifier = 5
    )
  TO
    SELECT query(
      SELECT * FROM NetworkElement ne2 WHERE ne2.identifier = 6
    )
  USING "DepthFirstSearch"
```

The generated output is an Agent AC that can be put on every NE in the network, because it starts traveling the network in a Depth-first fashion until it finds NE 5. Once arrived at NE 5, it will try to disconnect the link to NE 6 and terminate.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:ac="ac">
  <ac:ApplicationComponent identifier="travelTest" revision="1">

    <functions name="main">

      <!-- Put some objects on the stack, the QUERY
           instruction marks the end with "NULL"
      -->
      <statements name="LABEL">
        <arguments type="String" value="loop" />
      </statements>

      <!--  After the next statement we are on
            another NE and have its ID on the stack
      -->
      <statements name="VISIT_NEXT" />
      <statements name="PUSH">
        <arguments type="Integer" value="5" />
      </statements>
```

```
22        <statements name="EQ" />
23
24        <!-- If the result of VISIT_NEXT equals -1
25             we are done traveling
26        -->
27        <statements name="IF_TRUE_GOTO">
28          <arguments type="String" value="end" />
29        </statements>
30
31        <statements name="JUMP">
32          <arguments type="String" value="loop" />
33        </statements>
34
35        <statements name="LABEL">
36          <arguments type="String" value="end" />
37        </statements>
38        <statements name="PUSH">
39          <arguments type="String" value="NE␣found,␣disconnecting" />
40        </statements>
41        <statements name="PRINT" />
42        <statements name="PUSH">
43          <arguments type="String" value="NULL" />
44        </statements>
45        <statements name="PUSH">
46          <arguments type="Integer" value="6" />
47        </statements>
48        <statements name="DISCONNECT" />
49      </functions>
50
51    </ac:ApplicationComponent>
52
53  </xmi:XMI>
```

The code on the lines 10-30 keeps looping until NE 5 is found. The VISIT_NEXT instruction groups the *serialize*, *move*, *deserialize* and *start* instructions, which together cause the AC to be suspended, moved to the next NE, determined by the DepthFirst-Search module. From there the loop starts over again. The lines 32-46 print some information and disconnect the link to NE 6.

Appendix C

Solution to Case Study 2

This case study was performed on a network topology as depicted in figure C.1. A route was created from NE F to NE A and NE B. To do this, the following CoNSoLe program was compiled:

```
CREATE Route
  FROM
    SELECT query(
      SELECT * FROM NetworkElement ne1 WHERE ne1.identifier = F
    )
  TO
    SELECT query(
      SELECT * FROM NetworkElement ne2 WHERE ne2.neighbors.size = 1
    )
  USING "DijkstraShortestPath"
```

In this case study, the compiler obtains a complete model of the network and calculates the shortesth path between NE F and NE A and NE B using the Dijkstra shortest path algorithm. Now it puts the following AC code on NE F:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:ac="ac">
  <ac:ApplicationComponent identifier="forwarder" revision="1">

    <handlers handlerType="MSG_RECEIVED"
      functionName="handleMessage" priority="5" />

    <functions name="main">
      <statements name="LABEL">
        <arguments type="String" value="loop" />
      </statements>
      <statements name="SLEEP" />
      <statements name="JUMP">
        <arguments type="String" value="loop" />
      </statements>
    </functions>

    <functions name="handleMessage">
      <statements name="POP">
```

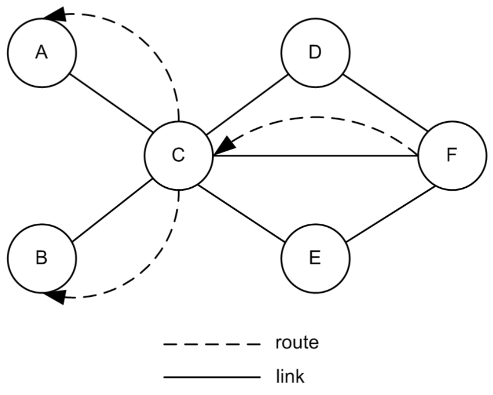

Figure C.1: *The network topology used for case study 2*

```
          <arguments type="String" value="packetDestination" />
        </statements>
        <statements name="POP">
          <arguments type="String" value="packetToken" />
        </statements>
        <statements name="POP">
          <arguments type="String" value="packetType" />
        </statements>

        <statements name="PUSH">
          <arguments type="String" value="packetToken" />
        </statements>
        <statements name="PUSH">
          <arguments type="String" value="1209804020" />
        </statements>
        <statements name="EQ" />

        <statements name="IF_TRUE_GOTO">
          <arguments type="String" value="match" />
        </statements>
        <statements name="RETURN" />

        <statements name="LABEL">
          <arguments type="String" value="match" />
        </statements>
        <statements name="PUSH">
          <arguments type="String" value="packetType" />
        </statements>
        <statements name="PUSH">
          <arguments type="String" value="packetToken" />
        </statements>
        <statements name="PUSH">
          <arguments type="String" value="C" />
        </statements>
        <statements name="PUSH">
          <arguments type="String" value="upvn" />
        </statements>
        <statements name="SEND" />
        <statements name="RETURN" />
```

```
60
61      </functions>
62
63    </ac:ApplicationComponent>
64  </xmi:XMI>
```

The above AC will be deployed on NE F. The lines 9-17 are the main loop, the AC is removed from the pool of active threads (SLEEP on line 13) if nothing happens. In case of an event, the AC will wake up and jump to line 19. At this point it will check if the packet contains the token 1209804020, which is a uniquely generated ID when the AC is compiled. This token is used to determine if a packet matches this particular route. If there is a match, the packet will be forwarded to NE C (line 43-58). Two similar ACs will be put on NE C. One forwards packets to NE A and the other forwards packets to NE B. This causes the stream of packets to be split into two streams at NE C.

Appendix D

Solution to Case Study 3

To implement this case study, we first checked if it was possible to create an AC. First we created a handler that responds to incoming packets (line 6-7 and 24-77). A second handler was created to flush the packet memory every 500 milliseconds (line 9-12 and 77-85). The *handleMessage* function assigns the different parts of the packet to variables using the *pop* instruction. Then it checks if the token of the packet is already in the packet memory, if so, the packet is discarded. Otherwise it is broadcasted. Because of time constraints, this case study was only implemented as ACs. We did not create rules in the compiler to map CoNSoLe statements to these kind of ACs.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3    xmlns:ac="ac">
4    <ac:ApplicationComponent identifier="forwardHold" revision="1">
5
6      <handlers handlerType="MSG_RECEIVED"
7        functionName="handleMessage" priority="5" />
8
9      <handlers handlerType="CLOCK_TICK" functionName="handleTick"
10       priority="5">
11       <arguments type="Integer" value="500" />
12     </handlers>
13
14     <functions name="main">
15       <statements name="LABEL">
16         <arguments type="String" value="loop" />
17       </statements>
18       <statements name="SLEEP" />
19       <statements name="JUMP">
20         <arguments type="String" value="loop" />
21       </statements>
22     </functions>
23
24     <functions name="handleMessage">
25       <statements name="POP">
26         <arguments type="String" value="packetDestination" />
27       </statements>
28       <statements name="POP">
29         <arguments type="String" value="packetToken" />
```

```
      </statements>
      <statements name="POP">
        <arguments type="String" value="packetType" />
      </statements>

      <statements name="PUSH">
        <arguments type="String" value="packetToken" />
      </statements>
      <statements name="CONTAINS" />

      <!--  If this token is new, store it and broadcast packet. Otherwise discard -->
      <statements name="IF_TRUE_GOTO">
        <arguments type="String" value="discard" />
      </statements>
      <statements name="PUSH">
        <arguments type="String"
          value="Token␣is␣new,␣storing..." />
      </statements>
      <statements name="PRINT" />

      <statements name="PUSH">
        <arguments type="String" value="packetToken" />
      </statements>
      <statements name="STORE">
        <arguments type="String" value="5" />
      </statements>

      <statements name="PUSH">
        <arguments type="String" value="packetType" />
      </statements>
      <statements name="PUSH">
        <arguments type="String" value="packetToken" />
      </statements>
      <statements name="PUSH">
        <arguments type="Integer" value="-1" />
      </statements>
      <statements name="PUSH">
        <arguments type="String" value="upvn" />
      </statements>
      <statements name="SEND" />

      <statements name="LABEL">
        <arguments type="String" value="discard" />
      </statements>
      <statements name="RETURN" />
    </functions>

    <functions name="handleTick">
      <statements name="PUSH">
        <arguments type="String"
          value="Resetting␣packet␣memory" />
      </statements>
      <statements name="PRINT" />
      <statements name="RESET_PACKETMEM" />
      <statements name="RETURN" />
    </functions>

  </ac:ApplicationComponent>
</xmi:XMI>
```

Appendix E

Solution to Case Study 4

Moving an Application Component

In the previous case studies we already used AC agents that travel over the network. In this appendix we will explain how the modules work that calculate which NE should be visited next by the Agent. Consider the following code fragment, taken from the Agent described in B:

```
  <statements name="VISIT_NEXT" />
  <statements name="PUSH">
    <arguments type="Integer" value="5" />
  </statements>
  <statements name="EQ" />
```

The VISIT_NEXT instruction is redirected to the DepthFirstSearch (DFS) module. This instructions invokes the *getNextDestination()* method in the module. The DFS module knows the AC and checks if it already stored some calculations in the module-Memory of the AC. If this is not the case, it initializes the collections *toVisit*, *visitedGray* and *visitedBlack*. Then it performs the normal DFS steps, discovering new neighbors, put them in the toVisit list and mark neighbors that have already been visited etc. After the DFS module has determed which NE should be visited next, it returns the identifier of this NE. Figure E.1 shows the structure of this explanation.

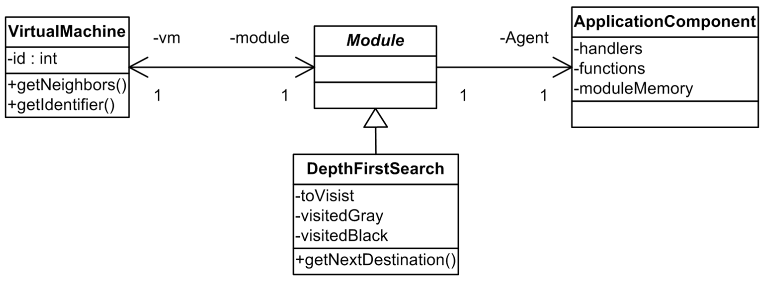

Figure E.1: *Structure of the DepthFirstSearch Module*

When the virtual machine receives the identifier of the NE that will be visited next, it pushes this value on the stack of the AC. Then it creates a packet, containing the thread of the AC and the AC itself. Before sending this packet to the next NE, it terminates the thread in which the AC was running. When the next NE receives the packet, another Agent recognizes the packet as being an agent and resumes the thread with the AC. From there the procedure starts over again.

Moving a Java Application

We also experimented to move and resume a normal Java application. This was possible, but the major drawback is the fact that the Java applications needs to manage its own memory and program counter. The following pseudo-code shows how this is done:

```
public static void main()
{
  if( resumeState() )
    restoreVariables();  // Restore previous state
    jumpToPreviousMethod();
  }
  else {
    initialize();  // This was the first time the application started
  }
}
```

The variables and program counter information were stored in an external XML file. A major drawback of this approach is the tedious work of doing your own administration of context information. Also, if this Java program would issue a CoNSoLe statement to a Network Component (NC) to move itself, something (either the NE, or an AC that runs in the Virtual Machine) must *know* of the existance of this type of Java applications. Also this 'something' must move the XML file that contains the context information of the Java application. A better solution would be support from the Java Virtual Machine to move suspended Java applications. Because of this, we only developed support for moving ACs.

Appendix F

Solution to Case Study 5

For this case study we used the scenario from case study 3 (Appendix C). We equipped every AC with the original source code. Then we introduced two new NEs: G and H. This is illustrated in figure F.1. Since a NE can only forward packets to its neighbors, the route between F and A and B has become invalid. We extended the ACs in the network with a handler and an additional function to detect and handle feedback packets:

```
1       <handlers handlerType="FEEDBACK"
2         functionName="handleFeedback" priority="5" />
3
4       <functions name="handleFeedback">
5         <statements name="PUSH">
6           <arguments type="String" value="sourceCode" />
7         </statements>
8         <statements name="COMPILE"/>
9       </functions>
```

The *sourceCode* variable (line 6) is a reserved variable that refers to the original CNS specification. The *compile* instruction will invoke the compiler. The compiler obtains a

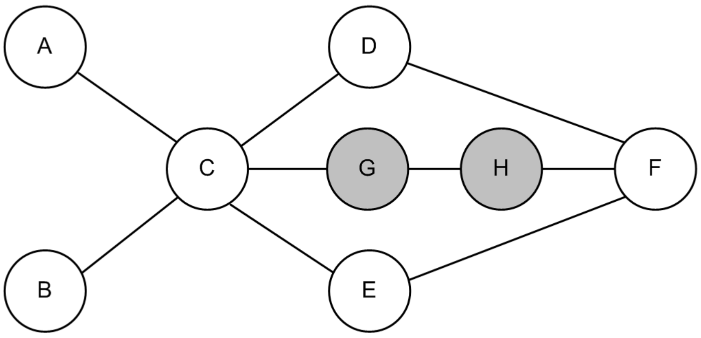

Figure F.1: *The network topology used for case study 5*

new model from the network, performs a new shortest path calculation and puts new ACs on NE F, E and C, replacing the old ACs.

Appendix G

Implementation of the Virtual Machine

Figure G.1 shows how the virtual machine, the compiler and the virtual network are implemented. All objects that reside in the "Java Virtual Machine" area are Java objects. The following scenario is depicted: A 'normal' Java application called "HelloWorld" contains zero or more CoNSoLe statements. The application also contains a NC (implemented using Java's Remote Method Invocation (RMI)). The CoNSoLe program is passed onto the NC which in turn passes it to the virtual network element. The Virtual Network Element passes it on to the compiler. The compiler is a Java object that acts as a bridge to the ATL transformation. The transformation receives the CoNSoLe program and returns one or more (textual) ACs. These textual ACs are wrapped into a Java object called "Ac" and are executed in the virtual machine.

Implementing support for Queries

Implementing support for queries has been done using the Hibernate Query Language (HQL) library. This library knows all instances of every Java object. Now the virtual machine can perform a query over the (virtual) network, because every (virtual) NetworkElement is actually a Java object.

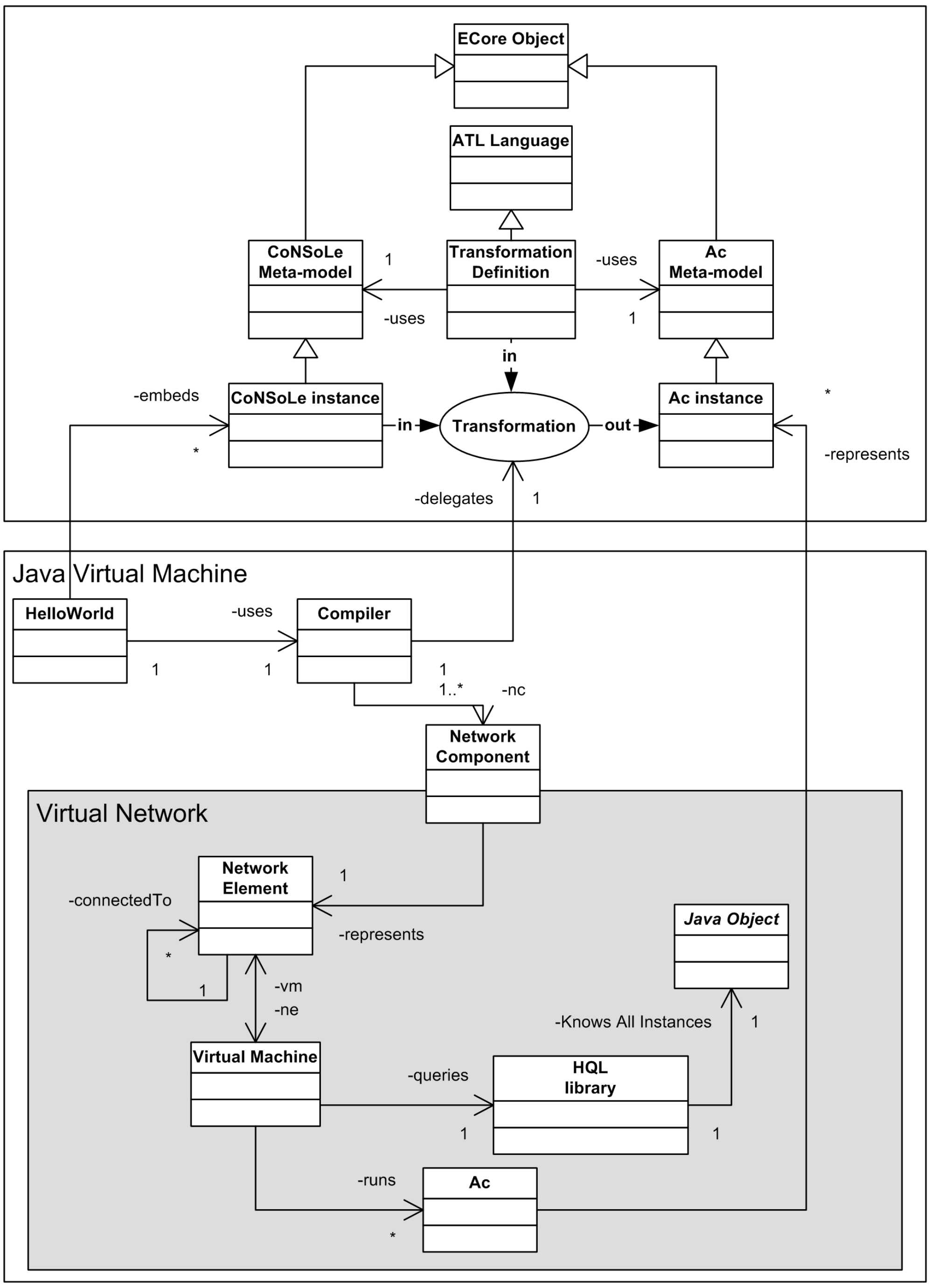

Figure G.1: *Implementation of the Virtual Machine, the compiler, and the virtual network elements*