UNIVERSITY OF OSLO
Department of Informatics

MICA: A Minimalistic, Component-Based Approach to Realization of Network Simulators and Emulators

Erek Göktürk

October 9, 2007

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo.
No. 653

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AiT e-dit AS, Oslo, 2007.

Produced in co-operation with Unipub AS.
The thesis is produced by Unipub AS merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Unipub AS is owned by
The University Foundation for Student Life (SiO)

Acknowledgements

This thesis wouldn't be possible without my supervisors Naci Akkøk and Ellen Munthe-Kaas. They have not only supported me on a professional level, but also on a personal level when needed.

The Distributed Multi-Media Systems research group have provided the cradle in which the ideas in these thesis have been formulated. I would like to thank Vera Goebel, Thomas Plagemann, Matti Siekkinen, Karl-André Skevik, Ovidiu Valentin Drugan, Tommy Gagnes, Matija Pužar, Norun Sanderson, Katrine Stemland Skjelsvik, Piotr Srebrny, Jarle Søberg, and Željko Vrba, with whom I have been able to discuss various topics at various times during my studies. Of these people, Matija Pužar deserves special credit for reading patiently through my articles, and Željko Vrba for many constructive discussion sessions that we had during the years that we have shared our office.

At the personal level, I would like to thank Çiğdem and Naci Akkøk, for hosting a young couple they hardly know in their home for two weeks when me and my wife arrived in Oslo. I would like to thank my dear wife Beyza for sticking with me through these four long years, and my parents-in-law for putting up with their son-in-law who seemed to have been indulged in strange complicated thoughts in his own little world. I also would like to thank my mother for her understanding, as she let me move two thousand kilometers away just after we lost my father.

For Beyza.

Contents

List of Figures

List of Tables

List of Abbreviations

ABI	Application Binary Interface
ACA	Autonomous Component Architecture
API	Application Programming Interface
ARP	Address Resolution Protocol
BOA	Basic Object Adapter
CB	Customized Behavior
CB	Component-Based
CCA	Common Component Architecture
CCM	CORBA Component Model
CCT	Critical Channel Traversing
CLS	Common Language Specification
CLSID	Class Identifier
COL	Component Object Library
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CU	Constructor Unit
CUI	Constructor Unit Instance
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interface
DIS	Distributed Interactive Simulation
DLL	Dynamic Link Library
DOM	Document Object Model
DSB	Dynamic Simulation Backplane
DSI	Dynamic Skeleton Interface
DSO	Dynamic Shared Object
DSR	Dynamic Source Routing
EJB	Enterprise JavaBeans
EU	Execution Unit
EM	Execution Manager
GUID	Globally Unique Identifier
HLA	High Level Architecture
IC	Integrated Circuit

ID	Identifier
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
JSP	Java Server Pages
JVM	Java Virtual Machine
KQML	Knowledge Query Manipulation Language
MAC	Media Access Control
MANET	Mobile Ad-hoc Network
MICA	Minimalistic Component-based Architecture
MIDL	Microsoft Interface Definition Language
MPI	Message Passing Interface
MPMD	Multiple Program Multiple Data
OLE	Object Linking and Embedding
OLSR	Optimized Link State Routing
OMG	Object Management Group
ORB	Object Request Broker
OSA	Open Simulation Architecture
POA	Portable Object Adapter
PVM	Parallel Virtual Machine
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTTI	Run-Time Type Information
SCM	Service Control Manager
SIDL	Scientific Interface Definition Language
SIFSUT	Stand-in for the System Under Test
SMF	Simulation Management Functionality
SPMD	Single Program Multiple Data
SUT	System Under Test
TCL	Tool Command Language
UM	Unit Model
UMI	Unit Model Interface
UML	User Mode Linux
UNO	Universal Network Objects
UNOIDL	Universal Network Objects Interface Definition Language
XML	Extensible Markup Language
XPCOM	Cross Platform Component Object Model
XPIDL	Cross Platform Interface Definition Language

Abstract

This thesis addresses the following question: What can be better ways of realizing network simulators and emulators? This question presents a moving target, since the meaning of "better" changes with changes in requirements due to rapid advancements in networking technology. As a result, this thesis is oriented towards identification of the design parameter space for constructing network simulators and emulators, with special focus on the software related challenges.

Network simulators and emulators have become major tools in experimental networking research because of several reasons, including the higher costs of experimenting on real networks. A simulation or an emulation-based experiment involves activities that can be categorized into the experimental design level or software level. While the experimental design level includes activities such as the identification of the system under test and construction of its model, the software level involves activities that relate to the construction of a simulator or an emulator as a software entity. At the software level, network simulators and emulators frequently require the network researcher to write the code for implementing the model elements used in the simulation. Since network researchers are not necessarily also experts in software, network simulators and emulators must have solid and simple software architectures that are designed to ease this burden of implementation.

Working towards formulating better ways of realizing network simulators and emulators necessitates a solid background about the available solutions in the field. Thus, this thesis presents surveys on network simulation and emulation, as well as a stance on the terminology on simulation, emulation, and testbeds.

In this thesis, a component-based approach is chosen as our approach to building simulators and emulators. The rationale for this choice is explained in detail, primarily for guiding further studies on software architectures for simulators and emulators. One of our main motivators in using components is to use the component-based structure of networks as perceived by network professionals, as a leverage in increasing the usability of simulators and decreasing their learning costs.

There are many design decisions involved in formulating a component-based approach. Making such decisions require a good understanding of the component concept and its use in software development, which we facilitate by covering some selected issues in component-based software engineering, and by presenting a survey of major component-based approaches with a special focus on their component models. This also helps us place the component-based architecture we have developed within the context of component-based software engineering. Furthermore, the design decisions involved in the development of our approach are documented thoroughly, since these decisions provide an idea about the range of decisions that need to be made in similar projects.

Our component-based approach is called MICA – The Minimalistic Component-Based Software Architecture. MICA aims to be minimal and easy to use. At the same time, MICA pro-

vides a detailed component model with several unique characteristics such as separation of worker and constructor components, and having a single control flow model from the perspective of the component developers, while being able to express multiprogramming, multitasking, and parallel processing through the use of component containers. MICA also includes necessary constructs for supporting transparent distribution of components. Furthermore, provided that certain conditions are satisfied by a simulator that is built using MICA, MICA allows model replacement and simulator interoperability at the software level.

We also describe two different component platforms that we implemented for the MICA component model. One of these platforms provides a single thread of control, therefore it is best suited for testing and debugging. Our other platform is built using the Parallel Virtual Machine (PVM) middleware, and it supports the whole range of control flow options defined in the component model.

We have also implemented a network emulator built using MICA, which is called DINEMO. DINEMO is a refactoring and extension of the NEMAN network emulator. It uses the TUN/TAP virtual network interfaces for hosting real applications distributed on multiple hosts in a simulated network. DINEMO acts as a proof-of-concept for MICA.

Chapter 1

Introductory Overview

In this first chapter, we will present the subject of this thesis, and an overview of its contents. The chapter will start with a discussion of what the guiding question for this thesis has been, and the motivation for asking that question. Following that, we will describe the contents of the thesis in the form of our contributions related to the research question. In the last section, we will briefly discuss what the research methods employed in this thesis are.

1.1 The Research Question and Motivation

The goal of this thesis is to gain insight into the answer to the following question: "What can be better ways of realizing network simulators and emulators?"

Why do we need to search for better ways of realizing network simulators and emulators? The indicators that pointed towards such a need at the outset of this study, were the following:

- Constant complaining in the networking research community about the learning time that is needed for using network simulators.
- Results reported in literature that lack a detailed list of model parameters. This observation was also pointed out by Perrone et al. [142]. Lack of inclusion of model parameters showed either a lack of understanding of how simulators work and the importance of the model parameters on the results, or an ignorance about it. Given the time needed for mastering network simulators and emulators, the attitude of the authors of these papers was probably a combination of both.
- Cavin et al.'s results [35], which point to the fact that results from different network simulators are hardly comparable.
- Our experience with NS-2 internals.

The question we have posed can be broken down into sub-questions. Such an analysis is needed not only in order to pave way towards a solution, but also in order to understand the question itself. We are going to organize our analysis of the question by focusing on three subjects: terminology, roles and preferred characteristics, and realization.

1.1.1 Terminology

The first group of questions that can be derived from the research question, are questions about the terminology: what exactly are those which we call simulation, emulation, simulator, emulator, and testbed? What do they mean in the context of networking research? The main function of these questions is setting the discourse.

The terminology is explored in Section 2.2. An important point turns out to be that whether a surrogate system is to be regarded as a simulator, an emulator, or a testbed depends heavily on the definition of the system under test in the experiment.

In Sections 2.5 and 2.6, simulation and emulation in the context of networking research are explored. Particularly, Section 2.6 and Appendix B present a brief but thorough survey, which has also been published in [73]. Furthermore, a revised version of [69] is included as Section 3.5, where an exploration of problems specific to emulation style experiments is presented.

1.1.2 Roles and Preferred Characteristics

The "realizing network simulators and emulators" part in our main question refers to production of simulators and emulators. Such production implicitly assumes the goal of producing "good" tools, which brings the question "What are preferred characteristics for a simulator or an emulator?" Since the attribute of being "good" is in the eye of the beholder, we looked into how the simulators and emulators are used by the network researchers. A result of this investigation was identification of two different roles: the experimenter, and the simulator or emulator developer. These two roles are easily confusable, and in fact are confused in networking research. NS-2 code base stands as a typical demonstration to such confusion. This subject is further discussed in Section 3.2, based on [71].

1.1.3 Realization

The last component of the main question, which is the part we have mainly focused on, is about "better ways of realizing". For this purpose, we attempted to create a software architecture that would have the following characteristics:

- Suggesting and supporting the easily confused developer-experimenter role division.
- Being minimalistic. By stripping the foundational layer to the bare fundamentals, it should be possible to increase the availability of the architecture for various extensions, and for use in re-factoring purposes.
- Supporting variety of different simulation and emulation techniques.
- Providing a composition and communication style, where submodels communicate in a way that is similar to how networking devices communicate in real networks. Such similarity is expected to reduce the learning time by leveraging the previous knowledge of the network researchers.

The initial design of the architecture was published in [70] under the name AMINES-HLA. The last form of the architecture, which is named as the Minimalistic Component-Based Software

Architecture (MICA), is presented in Chapter 4. In addition to the characteristics above, the MICA architecture further supports simulation interoperability and model replacement, as demonstrated in the scenarios presented in Sections 4.4.3 and 4.4.4. Earlier work on these scenarios were published in [72].

In order to test and demonstrate the MICA architecture, a monolithic mobile ad-hoc network emulator called NEMAN [149] was componentized on MICA and made distributable. The resulting emulator, called DINEMO, is presented in Chapter 5, and was published in [74]. Since our objective in building DINEMO is mainly the demonstration and testing of MICA, we have focused on the benefits provided through the use of MICA, and not on wide-scale promotion of DINEMO as a network emulation system. Such an effort is considered to be out of the scope of this thesis, since the synthesis of the MICA architecture required considerable resources for an understanding of component-based software technology, a wide background survey of network simulation and emulation, and the development of the component model for MICA and the platform implementations. Therefore the focus of this thesis has mainly been on the relationship between the software engineering aspects of simulator development, and the domain of networking. An implementation of a network simulator or emulator for wide-spread use, and promotion of DINEMO along with its further development, are left as possible future work.

1.2 Overview of Contributions

Here we shortly describe the contributions of this thesis, categorized according to the main research areas contributed.

1.2.1 Contributions to Experimental Networking Research

The contributions in the experimental networking area are the following:

- Through the short and pragmatic treatise of simulation, emulation, and testbeds, we provide a clear stance about the terminology used in experimental research in networking. Especially, we contribute by our discussion of what emulation is, and what makes emulation-based experimentation different from testbed-based experimentation. (Section 2.2)

- We present a detailed survey of network emulators reported in literature, covering forty two systems, and identify some more using which the survey can be extended. Through the survey, the main approaches and techniques used in systems for emulation-based network experimentation are identified. (Section 2.6 and Appendix B)

- We identify three different conceptual levels where the concept of components is being employed in the context of simulation and emulation systems: components of the system under test, model components, and simulator components. Differentiation of these levels helps understanding what are referred to as components in various systems reported in the literature and explains why the various works that claim to have a component-based approach appear to be referring to unrelated conceptualizations of components. (Section 2.4.2)

- We identify and discuss problems of emulation-based experimentation and emulators, which are not found in simulation-based experimentation and simulators used in experimental research in networking. (Section 3.5)
- We analyze stakeholders in simulation and emulation-based experimental research in networking. As a result, we identify two major roles, experimenter and developer, with conflicting goals. The confusion of these roles lead to poor structure and manageability of the simulator or emulator as a software product. (Section 3.2)
- We define a new software architecture that is specifically targeted towards realization of network simulators and emulators (Chapter 4). This new software architecture
  - supports the separation of experimenter and developer roles identified in the aforementioned analysis of stakeholders in simulation and emulation-based experimental research,
  - introduces a component model that aims to align the use of the concept of components at three different conceptual levels mentioned above,
  - supports simulator interoperability, model replacement, model implementation reuse, and run-time control over the run-time representation of the model being simulated from outside of the simulator. (Section 4.4)
- We present a component-based, extensible, and distributable emulator that uses TUN/TAP virtual interfaces, named DINEMO, for emulation-based experimentation where use of real protocol stacks and programs over simulated networks is needed. (Chapter 5)

1.2.2 Contributions to Component-Based Software Engineering

The contributions in the component-based software engineering area are the following:

- A new component model (Chapter 4) that
  - is minimal, in the sense that any element in the model is clearly targeted towards satisfaction of a set of defined goals for the model, and redundancies are minimized,
  - is detailed enough to be supported by multiple component platforms that are based on different technologies, but which can still run the same components without the need for re-compilation,
  - provides transparent distribution from the point of view of developers of the components of a certain type defined in the component model,
  - provides a control flow model that explicitly includes multiprogramming, multitasking, and parallel processing support.
- A stance on some terms that are sometimes used with vague meanings in the component-based software engineering literature. This stance serves to provide clean definitions for the terms component model, component platform, component-based architecture, and component-based framework. (Section 2.3.3)

- A survey of major component-based approaches, including OMG's CCM, JavaBeans, EJB, COM, Fractal, and more, with a special focus on their component models. In addition, a comparative presentation of the surveyed component models along with the model defined in this thesis in terms of control flow, lifetime management, component communication, and composition. (Appendix C and Section 6.1)

1.3 Research Method

This thesis is not intended to provide a definitive answer to the research question presented in Section 1.1. The question is dynamic: the meaning of the definition of "better" eludes any static definition, because the requirements of the experimental network research is changing rapidly with the changing technology. Two good examples on how the changes in technology introduced new requirements for the network simulators and emulators can be observed through the simulation studies for huge networks such as the Internet, and simulation and emulation studies for mobile ad hoc networks.

In this thesis, we take an "exploratory" stance towards the research question. Instead of setting our goal as "the ultimate! final! best!" network simulator or emulator, we aim to understand how simulators are built and can be built, from a software engineering perspective. This we do in order to be able to build new simulators and emulators as requirements change due to new needs arising.

Akkøk's PhD thesis [4] has a very good overview of research methods related to computer science, covering both the ACM taxonomy of subject areas, and another categorization that is based on a list of 14+2 subjects put together from works of Galliers and others.

With respect to ACM taxonomy of computer science, as presented in [4], the paradigm of this thesis is design. In this thesis,

- the requirements for software engineering of network simulators and emulators are examined in Chapters 2, and 3,
- a new component-based approach, called MICA, is specified and implemented as reported in Chapter 4, building upon the wide overview of the component-based software engineering approach presented in Chapter 2,
- and the new approach is tested by implementing a distributed network emulator, called DINEMO, as reported in Chapter 5.

The research methods from the classification of 14+2 subjects in [4], that are relevant to this thesis are the following:

"Surveys" approach involves searching for previous and relevant works in the literature. The surveys in this thesis are presented in Appendix B, Appendix C, and partly in Section 2.6 and Chapter 6.

"Subjective/argumentative research" is based upon opinion, speculation, et cetera, but is creative and provides insight. It is useful for building a theory or understanding. This approach can be observed in Chapter 2, and in the parts of Chapter 4 where we discuss the motivation and requirements for the MICA software architecture.

"Engineering" is development of technology, which involves construction, and reconstruction of systems. The Chapters 4 and 5 are most relevant to this approach.

"Grounded theory" approach lets relevant information or theory to "emerge" from studies and observations, as opposed to organizing a theory at the outset. In this thesis, we have draw upon our surveys to formulate the stances that are presented in Sections 2.2 and 2.3.3, and the identification and separation of the different levels that the concept of component is used in simulation, which is presented in Section 2.4.

Chapter 2

Background

In this chapter, we will provide some information and observations that form the background for this thesis. We start by describing the research area, since it determines what subjects are related to this thesis. Then, we will present our stance on simulation, emulation, and testbeds, which forms a terminological anchor point for further discussions. In Section 2.3, we will cover some issues related to component-based software. This section provides a background that is more or less independent of specific component-based systems or development approaches in the literature. Various component models that exist today are briefly discussed in Appendix C. In Section 2.4, we have a look at how the component-based approach is perceived and used in the simulation domain. Then we leave the component-based approaches aside for a moment to look into network simulation in Section 2.5, and network emulation in Section 2.6. A more detailed survey on network emulators and testbeds can be found in Appendix B. Finally, we look into component-based network simulation and emulation in the last section.

2.1 The Research Area

The research area that subsumes the topic of this thesis is component-based engineering of network simulators and emulators. This area lies in the intersection of two major research areas: simulation, and software engineering.

Simulation is a large and interdisciplinary field, which involves modeling, mathematics and statistics, and computing science. The simulation researcher needs modeling in order to represent the phenomena to be simulated. Mathematics and statistics are needed for experiment design, modeling, and in order to manage the uncertainties. Computing science provides the methods for computing the outcome of the model for a set of given inputs. In addition, application of simulation in some domain naturally necessitates a good understanding of the domain.

Networking is one such application domain where simulations are used frequently as one of the major tools. The advent of large scale networks and wireless communications in the networking domain have resulted in new challenges for research in simulation. At the modeling level, scalability and multi-resolution modeling are issues of concern. Today, there are three major approaches to network modeling: packet-level event based approach, flow based approach with event based simulation or partial differential equation solvers, or hybrid approaches where packet-level and flow based approaches are used together.

Emulation can be defined as using a simulator in an experimental setting where it has to interact

with non-simulated entities that can be considered as part of the phenomena under study. Network emulation is a popular experimentation technique in networking research, because it allows researchers to test real protocol implementations, applications, or hosts in situations that might not be possible or feasible to realize in the real networks.

The models of networks are constructed according to the architecture of some chosen network simulator, and using the simulation semantics supported by this simulator. In other words, the meta-model to be used by a network researcher in constructing the model to be simulated in his/her studies, is determined and imposed by the simulator to be used. Many network researchers come up with their own simulator for their experiments, in order to be able to control the complexity of the meta-model used. This suggests that the complexity of the meta-models used by the existing network simulators is prohibitively or discouragingly high. As a result, there exist today a high number of experiment specific or researcher/research group specific network simulators.

While there is relatively more interest on the meta-model imposed by the network simulators, there has been less work focusing on engineering the simulator's code. The reason seems to be the conflicting goals of stakeholder roles, namely experimenter and developer, in network simulation: the network researcher focuses on realizing an experiment, not necessarily on implementing better tools. Due to special needs of network simulation, where the models are generally to be implemented by the network researcher, implementing better tools also involves providing ways of implementing tools in a better way. The subject of this thesis from the networking point of view, addresses this need for work that is focused on ways of implementing simulators in better ways, as tools for the network researchers.

Since simulators are software, software engineering naturally arises as the discipline to look for the answers to the question of what would be better ways of implementing simulators. What we are especially interested in is how to implement simulators in a way that is easy to understand for the network researchers, that easy composition of model implementations is supported, and that is extensible to support different simulator implementation approaches or different meta-models. Composition of the model to be simulated in terms of smaller models, which might be implemented by different network researchers, provides the natural lines through which modularization of the code can be realized. Additionally, if a simulator is to be used in an emulation experiment, there has to be a representation of real entities in the stand-in for the system under test in the experiment. These submodels and real entities carry the characteristics of a "component", with regard to the way this term is used in component-based software engineering. Therefore, a component-based approach appears to be the most relevant approach for this thesis.

However, existing component-based approaches do not necessarily have the same set of goals as we have in this thesis. The shortcomings of the existing approaches are related to complex services, limited extensibility of control flow, and limited support for fine granularity. A survey of existing component models is presented in Appendix C, and a comparison to our approach is presented in Chapter 6.

While our focus in this thesis has been network simulators and emulators, we are aware that the approach in this thesis might be applicable beyond this particular domain. However, such a claim is avoided, since it might have made this thesis intractable. Further exploration of how the approach in this thesis can be carried on to other domains for simulation applications, or when it can be useful in developing applications other than simulators, is left as future work.

C. Computer Systems Organization
- **C.2.** Computer Communication Networks
  - **C.2.0.** General: Experimental studies on all issues related to computer communication networks fall under the potential application domain of this thesis.

D. Software
- **D.2.** Software Engineering
  - **D.2.11.** Software Architectures: Includes domain specific architectures and patterns
  - **D.2.13.** Reusable Software: Includes reusable libraries
- **D.4.** Operating Systems
  - **D.4.7.** Organization and Design: Includes distributed systems

I. Computing Methodologies
- **I.6.** Simulation and Modeling
  - **I.6.3.** Applications: Since DINEMO is an application of simulation in the emulation–based experimental research in networking
  - **I.6.5.** Model development: Includes modeling methodologies
  - **I.6.7.** Simulation Support Systems: Includes environments

Figure 2.1: Main subjects of this thesis according to 1998 ACM Computing Classification System.

D. Software
- **D.2.** Software Engineering
  - **D.2.2.** Design Tools and Techniques: Includes modules and techniques, and object–oriented design methods
  - **D.2.3.** Coding Tools and Techniques: Includes structured programming, and object–oriented programming
  - **D.2.12.** Interoperability: Includes distributed objects
- **D.4.** Operating Systems
  - **D.4.1.** Process Management: Includes scheduling, and multiprogramming, multitasking, and multiprocessing

Figure 2.2: Additional subjects, which are also related to this thesis, but more remotely.

2.1.1 According to the 1998 ACM Computing Classification System

The main subjects in computer science that subsume the research question of this thesis, according to the 1998 ACM Computing Classification System (CCS), are given in Figure 2.1. Additionally, the subjects in Figure 2.2 also stand as related.

2.2 A Stance on Simulation, Emulation, and Testbeds

In this section, we will elaborate on simulation, emulation, and testbeds, in order to clarify our stance on how we perceive these related but different concepts. This discussion serves to prepare the grounds for the construction of our approach, and sets up the terminological space for our research.

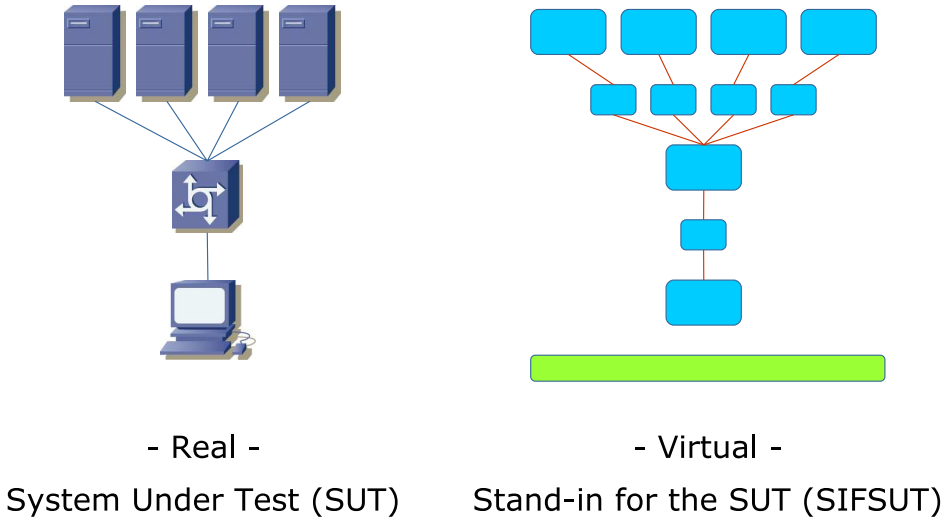

Figure 2.3: SUT and SIFSUT in simulation. The example system shown on the left is the SUT, which is real, and the SIFSUT on the right is completely virtual.

2.2.1 Simulation

Simulation is an old and intriguing idea, even from a philosophical point of view. In essence, *simulation* is the use of a surrogate system for obtaining information about the original system. We will refer to the original system as the *system under test* (SUT), and the surrogate system as the *stand-in for the system under test* (SIFSUT)[1] (see Figure 2.3). A *simulator* includes a SIFSUT, and not necessarily but generally parts that act as the surrogate system for the context of the SUT in situations of interest[2].

Computer simulation is the use of a mathematical system built using computational methods as the simulator in a simulation. When defined as such, the history of computer simulation actually precedes the history of electronic computers. Before the electronic computers were built, there were the so-called human-computers, the computing personnel who carried out computations and algorithms by hand. Some of the work done by human-computers, can also be considered as computer simulation.

The SIFSUT in computer simulation is a combination of two parts: the model of the SUT, and methods to computationally process the model in order for the SIFSUT to be able to act sufficiently similar to the SUT it stands for. We will frequently refer to the former simply as the *model*, and the latter as the *simulation management functionality (SMF)*. These two parts are glued by *meta-models*, which describe how model structure and semantics relate different methods used as SMF.

[1] We will use "SUTs" and "SIFSUTs" to mean "systems under test" and "stand-ins for systems under test", respectively.

[2] A precise definition of system, the relationship of being a surrogate, and the epistemological status of the information gathered through the virtual, are subjects of philosophical debate [27, 28, 59, 119, 167]. Note that since our intention here is not one of providing a full fledged philosophical discussion, we will use the terms system, surrogate, and information somewhat loosely.

Considering this partitioning, the field of computer simulation focuses on the following tasks and questions:

- How can the model be constructed? What methods of model construction would end up in SIFSUTs that sufficiently reflect the SUT being modeled? The development and verification of mathematical and knowledge engineering tool sets and theories for constructing models. This focus is sometimes also referred to as *conceptual modeling*.
- Identification, abstraction, classification, and study of methods to computationally process the models. Given a model, these methods are used in construction of the SIFSUT. For example: the study of discrete-event systems, and ensuring causality in parallel discrete event simulation (PDES).

In computer simulation, the term simulator is more frequently used for referring to software environment that includes implementation of a set of methods for processing models. Coupled with suitably expressed models, these environments become a SIFSUT, or a simulator as we have defined previously. As the borders between models and the computational processes that work on these models become somewhat blurred, this double meaning of the term simulator gets confusing. In this thesis, we will try to avoid this dual usage by referring to such software environments as simulation systems or simulation libraries.

A *simulator library* is a library of model implementations. Simulator libraries are also frequently called *model libraries*, but we will prefer the former in order to stress that the representation of the models is at the implementation level. At this level, each of these models can themselves be regarded as a SIFSUT or a simulator.

The models in a simulator library share a meta-model or some meta-model characteristics, in such a way that they can be composed into a simulator. By and large, the shared meta-model or characteristics are not documented explicitly, but only expressed through a fixed SMF using which the model implementations should be run.

A *simulation system* includes a fixed SMF implementation. Generally, simulation systems come with a simulator library. In some cases, a simulation system is built by a simulator developer by choosing and composing parts from a library for implementing the SMF. We will refer to such a library as a *simulation library*. Regardless of whether a simulation system or a library is used, the resulting simulator includes an SMF implementation that requires to be instructed by a script or a small size program for building the model to be simulated from models in one or more simulator libraries.

As it is apparent from its definition, the application field for simulation covers more or less everything. Applicability is perhaps only limited by the feasibility of resource availability. In one form or another, computer simulations are developed and used by all engineering fields, management and finance, natural sciences, and teaching and training in diverse fields from aeronautics to military to medicine. Therefore today, there is no shortage of simulation conferences, and most of them include various fields of applications as track subjects.

2.2.2 Testbeds

Before we move on to discuss what emulation is, we should first define testbeds, since they lie on the opposite end of the spectrum from the simulation.

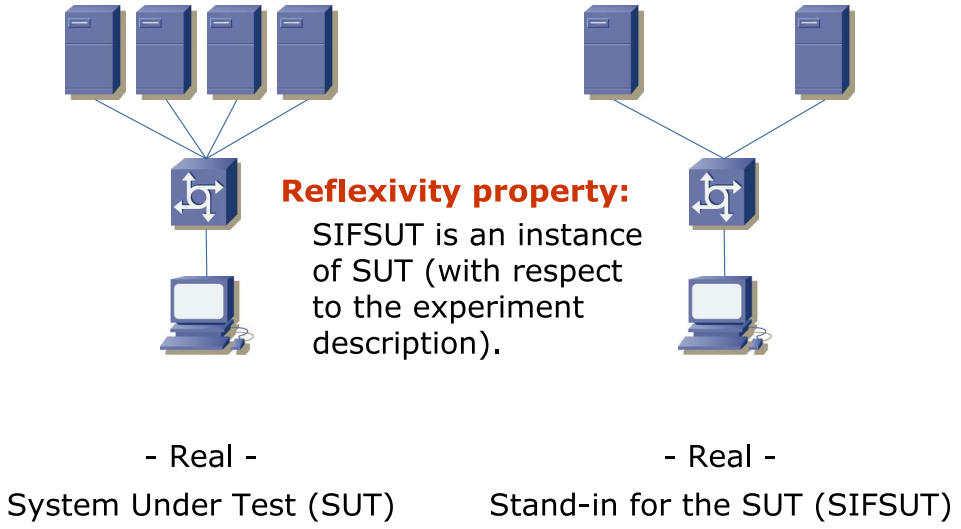

Figure 2.4: SUT and SIFSUT in testbeds. The example system shown on the left is the SUT, which is real, and the SIFSUT on the right is also real and is a subsystem of the SUT.

The distinguishing property of a testbed is that it satisfies what we will call the *reflexivity property*[3]: a testbed itself is a perfectly normal instance of the system that is under study (SUT) in a particular experiment, which is used for meeting various experimental objectives such as collecting data to be interpreted for obtaining indicative results about the SUT. The testbed might be a smaller scale example of the same kind of system as the SUT, but it might not be an all-aspects-scaled-down version of it. For example, a token ring network of five nodes can be used for developing an application targeted for token ring networks. Then, the application would later be deployed onto a token ring network of hundred nodes. Another simple example is shown in Figure 2.4.

In any testbed-based experiment, one should carefully evaluate in which dimensions the test results can be regarded as indicative for the properties of the SUT. For example, scalability is one of the dimensions testbeds are seldom useful for deriving indicative results about. Consider the example introduced in the previous paragraph: using only five hosts, whether any protocols will scale to networks of bigger size cannot be determined unless one can replicate the situations of interest in the big target network using just five nodes.

Examples of testbeds in experimental networking research can be found in Appendix B.

2.2.3 Emulation

In an emulation-based experimental setup, some of the entities in the SUT and its relevant context are represented by surrogate systems, while at least one entity is used as real (see Figure 2.5). Therefore, it is more likely in emulations than in simulations, that the level of abstraction will differ between different parts of the experimental setup.

[3]Olivier Dalle was the one to put a name to this property when the author was explaining the concept to him in private conversations.

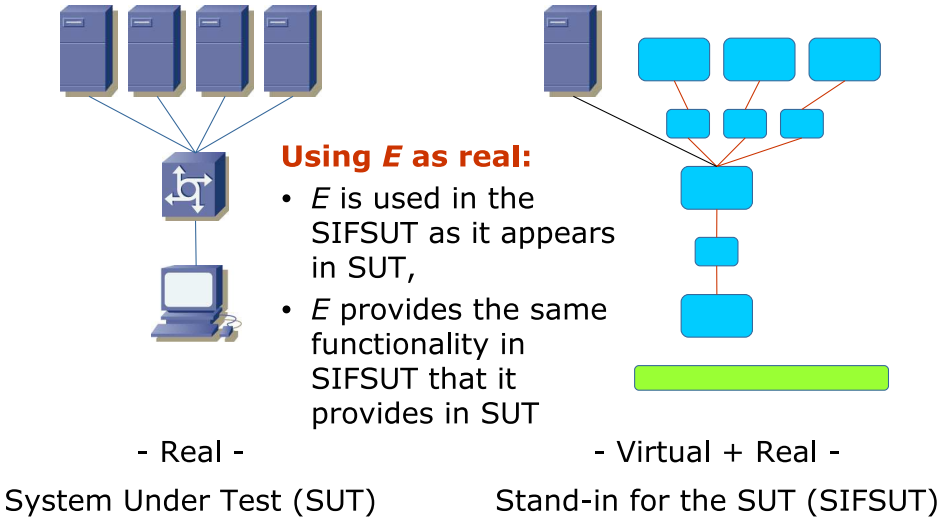

Figure 2.5: SUT and SIFSUT in emulation. The example system shown on the left is the SUT, which is real, and the SIFSUT on the right is a combination of virtual and real.

Providing a philosophically sound definition of what it means to be used as real in the experimental setup is not a straightforward task. Nevertheless, we will define the *property of being used as real* as follows: in an experiment, an entity E in the SUT or its relevant context is considered to be used as real, if in the experimental setup

- E is used as it appears in the SUT or its context,
- E is in relation with the representations of only those entities it would be in relation with in the SUT or its context,
- and E provides the same function it would provide in the SUT or its context.

The question to ask in order to find out whether an experimental setup is emulation-based or testbed-based is this: what is simulated in this experimental setup? The answer depends not only on the properties of the experimental setup, but also on the definition of the SUT and its context in the particular experiment. The same setup might be considered as a testbed for one experiment, and as an emulation for some other experiment. This creates a considerable confusion about the difference between testbeds and emulation-based experimental setups.

It is now time to present our definition of emulator. Let S be a subset of the entities in the SUT or its relevant context in an experiment. Furthermore, let S_r be the set of entities in the SUT or its context that interact with the entities in S. Then, a simulator that represent S in the experimental setup is called an *emulator* if

- the entities in a non-empty subset E_r of S_r are used as real in the experimental setup,

- the simulator is capable of interacting with the entities in E_r using the same modes of interaction that the entities in S would have used when interacting with them in the SUT or its context.

This definition can be demonstrated using flight simulators as an example. The so called flight simulators can be considered as emulators, if one defines an experiment that considers the pilot as part of the SUT or its context. A flight simulator acts as the representation of the plane and its surroundings. It interacts with the cockpit personnel using the same modes of interaction that they would experience in a real situation, except for a few effects such as gravity. Therefore, for a suitably defined experiment, a flight simulator is not a simulator but an emulator. This logic applies to many other systems used for educational purposes such as in surgeon training.

Consider also the software applications that act as if they are a particular type of machine or operating system. In fact, these are popularly called "emulators". For example, there exists "emulators" of Commodore Amiga, ZX-Spectrum, and Atari platforms, and for operating systems like Palm Os.

Time is an interesting entity that is an important part of the SUT and its context in many experiments. Executing in a way that synchronizes the real time (a.k.a. wall-clock time) with the time in the simulated system (a.k.a. virtual time), is considered to be a characteristic property of emulators by some researchers. Actually, time is only another part of the reality that in theory can be simulated in an emulation-based experimental setup. A simple example is provided by the various computer system emulators mentioned in the previous paragraph. In the SUTs of these emulators, time is already abstracted as timing signals. Using this abstraction, these emulators can be made to run slower or faster than real-time, while still carrying the properties of an emulator since they continue to interact with (run) real programs.

2.3 Component-Based Software

Component, "software component" to be more precise, is an elusive concept. There are many definitions in the literature, stressing many different characteristics attributed to components. In fact, it is in itself an intriguing question why the concept of software components is an elusive one. Is it simply because there is a considerable population of developers today, almost each of which has a perspective of their own? But then, why is it so? Does this point to an immaturity in the field?

Perhaps, some of the answers lie in the fact that there exists a self-similarity relation between the software and its components: a software component, which is a component of a software, is itself a software. Continuing to think along this line, we arrive at two questions that must be studied philosophically, or to be more precise ontologically: what is software, and what does it mean to be a component? Hardware, which has traditionally been the complement of software, is now being "coded" using languages not very different than those that are being used to code software. Therefore, we appear to be navigating philosophically near-blind, given the relatively little amount of work done or in progress towards understanding the ontological status of software itself. One exemplary attempt at such a philosophical exploration is given by Weber [187], based on Mario Bunge's ontological approach [27, 28].

However blind we philosophically might be, the motivation behind organizing software as components is a simple one. It is generally conceived that the products of the other "more mature" en-

gineering disciplines are constructed out of more or less standardized components. Availability of such a concept of components in the discipline provides the practitioners with a choice of focusing either on studying component making, or on building systems by putting together available components in systematic ways. Wide use of such components also creates a stable market for them, which becomes a driving force for increased production and revenues in the discipline under certain conditions.

It was as early as 1968 [122] when the idea of software components first found their way into the literature. It appeared in the form of the metaphor of software ICs (Integrated Circuits), with a special reference to systematicity. As a matter of fact, many ideas about components are also to be found in the object-oriented programming approach, work on which started with the SIMULA I project in 1961 by Kristen Nygaard and Ole-Johan Dahl [131].

In the mid-80s, the object-oriented approach started to gain serious momentum. In addition to C++, which is currently very popular, one of the important developments of that time was Objective C. In the context of his work on Objective C, Cox points to the different architectural layers in which objects can be used, and compare these layers to gate, block, chip, and card level components in electrical engineering [42]. Furthermore, although elements in all these layers provide encapsulation, components become less tightly-coupled as one goes from gate level to card level. Therefore, Cox concludes that although tightly coupled objects are necessary, there is a need for another abstraction layer where the integration of a software system can be made using loosely coupled "higher-level" objects.

Component-based approaches have become quite popular since mid-90s. Today, there exist various different component models and platforms. A survey of these component models is provided in Appendix C, which presents a short overview of OMG's CORBA Component Model, Sun's JavaBeans and Enterprise JavaBeans, Microsoft's COM and .NET, Common Component Architecture, Fractal Component Model, Autonomous Component Architecture, Mozilla's XPCOM, Universal Network Objects, and some others.

In this section, we will shortly discuss some of the issues and concepts related to component-based approach. It is not our intention to provide a comprehensive discussion of all issues related to component-based approach. Instead, the subjects discussed below are chosen for their relevance to this thesis.

2.3.1 Categorical Definition of Component: Characteristics

In this section, we will present a brief overview of different definitions provided for "component" in the literature. One of the major resources for this section was the excellent book authored by Clemens Szyperski [173], with contributions from Dominik Gruntz and Stephan Murer. Throughout this book, Szyperski provides several definitions, drawing upon a set of characteristics he attributes to the software components:

- being units of independent production, independent acquisition, and independent deployment,
- being units of composition that have contractually specified interfaces, and with all context dependencies declared explicitly,
- being executable units which can be composed into a functioning system.

Szyperski defines "component" categorically through these characteristics. In fact, he is not alone in preferring a categorical definition: a declarative definition of "component" is usually avoided. In Chapter 11 of his book [173], Szyperski quotes various definitions by others. These definitions appear to be clustered near the two opposite ends of a spectrum, depending on what properties they stress. At one end, the definitions focus more on implementation or architectural organization of software using components, by referring to persistence, incoming and outgoing interfaces, plug-and-play capability, coherence, coupling, and collaboration and interoperation [20, 52, 80, 83, 89, 129, 133, 136]. In contrast, at the other end of the spectrum lie life-cycle related concepts such as being prefabricated, being pretested, reuse, maintenance, deployment, support, and independent development [52][4]. Concerns that are orthogonal to this spectrum also creep into the discussion, such as being business process aligned [83, 190].

From our perspective, the main characteristics of components appear to be the following:

Independent development: Components are independently developed, possibly by different developers. They are packed as prefabricated and pretested units of deployment. They are also maintained and supported independently.

Independent deployment: Components are units of deployment: they cannot be partially deployed. Furthermore, they are usually deployed by third party. Therefore, they have to be self-contained.

Easy composability: Components are for composing. Such composition is likely to be realized in the context of other components developed independently. Therefore components need to be designed for late integration.

Coherence: Components are coherent units of software. They exist at a level of abstraction where they "directly mean something to the deploying client" [173].

Explicit context dependencies: In order to be both composable and decoupled, components have to define explicitly what they require from the deployment environment and other components. Naturally, they also should declare what they provide. Such dependencies are made explicit through *contracts*, which include "incoming" and "outgoing", or "requires" and "provides" interface definitions. Contracts provide a mutual understanding among different developers, and among developers of components and those who do the deployment. For this purpose, a contract should specify how a particular component can be deployed, how it can be instantiated once deployed (and installed), and how the instances behave through the advertised interfaces.

Up to this point, we have focused on the characteristics of components. In addition, one can also investigate characteristics that emerge in systems due to being constructed using a component-based approach. According to Szyperski, the component-based approaches in software engineering aim to build systems that exhibit the system-level characteristic called *independent extensibility*: a system is independently extensible, if it can cope with the late addition of extensions, which are components to be plugged into the running system when needed, without requiring a global integrity check [172].

[4] Szyperski also cites a whitepaper by the Meta Group which was acquired by Gartner Inc. in 2005, Jed Harris as quoted by Orfali et al. in [136], and a report by Ovum on distributed objects published in 1995.

2.3.2 Components, Component Instances, and State

The relationship between the components and component instances can be compared to the relation between classes and objects in object-orientation. A *component instance* encapsulates a state, and contributes to the functionality of the system it is composed into, by collaborating with other component instances through the set of interfaces it provides, and interfaces of other component instances that it uses. Component instances have their own identity, even when they can only be instantiated following the singleton pattern [66] in a system. A component is a description of the component instances, from which instantiation can be made. In general, similar to how object-orientation is supported at run-time, the behavior defined in code in a component is shared either as the source executable, or as memory fragments, among different instances of the component.

Meta-classes or templates in the object-oriented world provide potential metaphors for explaining what it means for a component to have a state. We define *component state* as the state associated with a component that determines the behavior and initial state of component instances created from the component. Component state is not modifiable by the instances of a component, and may even be inaccessible to them. A definition of state of a component instance can be easily constructed using the similarity of component instances with objects in object-oriented approach.

These definitions relate to a characteristic attributed to components by Szyperski: having "no (externally) observable state" ([173], pp. 36). We are not in full agreement that this is a characteristic of components, as discussed in the brief critique of this characteristic in the Appendix A of this thesis.

2.3.3 The Four Components of the Component-Based Approach to Software Engineering

Component Model

A *component model* is a description of what it means to be a component. It defines what a component is, how a component can interact with other components, what a component's execution and data flow characteristics can be, how components can be composed, and so on. This is the most abstract level view of components: a component model is the model of components.

As an example, MICA, which is presented in this thesis in Chapter 4, describes a component model that we propose to serve as part of software architectures for network simulators. In Appendix C, we discuss more about component-based approaches from the point of their component models.

Our definition of the component model relates to the discussion of the concept by Heineman and Council [80, 173]: a component model specifies the type of explicit context dependencies components may have (interaction standards), and how a set of components can be composed (composition standards). Therefore, a component model describes partly how a component is constructed (not its production, but its structure).

Component Platform

A component platform is a computational environment on which components as defined by a particular component model can be implemented and run. Sometimes, a component platform is also referred to as a run-time infrastructure or environment. There may exist multiple component

platform implementations for a component model. As component models provide different levels of detail about what a component is and its interactions with its environment, different component platforms for some component model may or may not be able to host a shared set of components.

CCM, CCA, Fractal, and MICA are examples where the component model is intentionally defined and described separately from the component platform(s). The main reason for such a separation is to allow component platforms that focus on different domains or that use different technologies to appear. Other component models, such as JavaBeans, EJB, COM, or .NET, have been developed as de facto standards by certain vendors, which also provide either the only or the reference component platform implementations. As an example to the level of detail in the component models, ACA leaves many design decisions to its component platform. Therefore, although the intention was not backing ACA up with a single component platform, it has been tightly coupled with the platform that was introduced with it. Fractal provides another example, since it leaves the issue of control flow to platform implementations. MICA will be presented in Chapter 4, and the other component models mentioned are shortly discussed in Appendix C.

Our definition of component platform agrees with Heineman and Council's definition of "component model implementation" as the dedicated set of executable software elements required to support the execution of components that conform to the model ([173], pp. 204). Szyperski appears to be referring to the same concept, in his definition of component platform ([173], pp. 549): "the foundation for component to be installed and execute on."

Component-Based Architecture

Component-based architecture is the set of types of components, in terms of what is being modeled, and the description of their relationships. Therefore, it identifies how the association will be established between the relevant entities in the target domain of the software being engineered, and components as defined by a chosen component model.

For example, if the software is an accounting system, the architectural elements may involve the data entry points, databases, and various reporting tools. The architecture might detail the elements, and how they are related, such as different kinds of account books which are part of the database.

Component-Based Framework

A component-based framework is a collection of rules and specifications[5] that define the interaction of components towards some clearly defined set of reusable design goals. A component-based framework restricts and guides the implementation of a component-based architecture for a particular domain. It adds more details up to the point where the system can be realized by deploying and instantiating suitable components on a chosen component platform.

In some component-based approaches, some services that are expected to be frequently used by the components are provided to the components as libraries. While these services are not implemented by other components in the system, they are nevertheless regarded as part of the framework. For example in CCM or EJB, component context may provide the component with services that are not necessarily provided by other components in the system.

[5]Such rules and specifications for a single component will be called its contract in the next section.

In one component-based system, multiple component-based frameworks may coexist, and one component may satisfy contracts from multiple frameworks. For example, one framework might be defining event distribution services, while another might be defining semantics of some particular kind of business transactions, and a component might be adhering to both. This style of combining frameworks can be regarded as horizontal. In vertical combination however, a framework may be implemented using other frameworks. This can even be established through composing components using one or more frameworks in order to build a composite component (see Section 2.3.6), which can be used in the implementation of a "higher-level" framework. Szyperski also points to the combination of frameworks by stating that "component frameworks are themselves components plugging into higher-tier component frameworks." ([173], pp. 548)

2.3.4 Interfaces

Interfaces are reified entry or exit ports of component instances for data or control flow. Behind these ports, there also exists a transportation system that carries data and control around. The characteristics of the transportation system are necessarily a part of the interface mechanisms defined in a component model.

There are many different ways of defining what interfaces are, and how they can be used by component instances. Some possible configurations that can be supported in different component models are shown in Figure 2.6.

Different categorizations of interfaces are being used in the community. One categorization suggests that one can identify or design for "provides" vs. "requires" interfaces. While "provides" interfaces specify how to access the services provided by the component instance, "requires" interfaces specify what services the component instance requires in order to fulfill its function. It should be noted that "provides" vs. "requires" categorization is based on intent of the designer, and does not seem to have solid support at the programming language level. Furthermore, it is possible to conceive designs that involve interfaces in which provided and required services are mixed. Provides and requires interfaces also appear to suggest a bidirectional and request-reply type of communication, which may necessitate more complex mechanisms in the transportation system.

"Incoming" vs. "outgoing" interfaces is another categorization. This categorization assumes a unidirectional data or control flow. An interface is an incoming interface if data or control is flowing towards the component instance. Otherwise it is an outgoing interface. A request-reply type of communication can be implemented using one incoming-outgoing interface pair for each side of the communication.

At the implementation level, two approaches to implementation of interfaces can be identified: procedural and object interfaces ([173], pp. 50). Procedural interfaces are composed of static call-points defined for a component or a component instance. Since procedural interfaces are static, a component or a component instance cannot have more than a single instance of such an interface. On the other hand, object interfaces can be dynamically created, destroyed, and passed on to other component instances. Object interfaces are like ambassadors acting as instances of a particular interface of a particular component instance.

Whether interfaces are used at both ends of the communication or not, and whether procedural or object type interfaces are used, are component model design decisions. However, these decisions determine the options available to the component platform developer in devising a method for

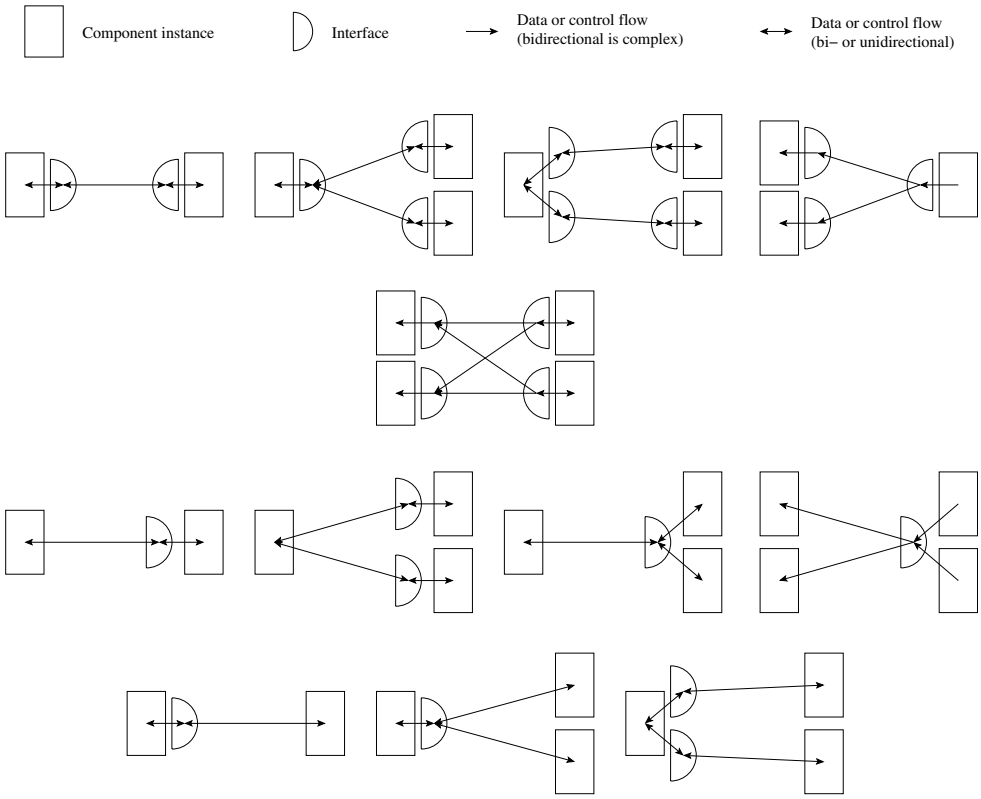

Figure 2.6: Various different approaches to formulating interfaces in component models. In all figures, initiating side is drawn on the right.

establishing the binding between the caller and the callee component instances in a way that keeps their coupling to a minimum.

2.3.5 Contracts

Contracts are documents that constrain the apparent behavior of a component. They serve as the agreement that the instances of a component should adhere to in their interactions with other component instances. Contracts may prescribe both functional aspects, such as syntax and semantics of an interface, and non-functional aspects, such as service level specifications. While the terms "contract" and "interface" are sometimes used interchangeably, in fact contracts subsume the definition of relevant interfaces. Especially, non-functional behaviors are not always explicitly represented in interface designs, while they should be included in contracts.

It is this author's opinion that contracts cannot be independent of component models. Nevertheless, contracts may strive to address several more-or-less compatible component models at the same time.

Contracts can also be used to guide the choice about the component platform to be used in a project, if more than one platforms are available to choose from. A design in terms of contracts

that is prepared early in the project development, may help formulating the criteria for the component platform choice. Such a set of contracts relates closely with the concept of component-based frameworks.

Testing and Verification

The independent extensibility property of component-based systems, which was introduced in Section 2.3.1, comes with its own problems. Since components are independently developed, tested, and deployed, a system level analysis for global integrity check is not very feasible at the time when the components are to be plugged into a running system. Even when the integration of components is not done while a system is running, a system level analysis is costly. Therefore, it is preferable to avoid such an analysis as much as possible.

If a component is not to be tested in the context of a system, what is it to be tested against? A system design can be expressed in terms of contracts, which then need to be satisfied by component instances at run-time. Therefore, contracts that a component is designed to satisfy provide the requirements against which to verify the component.

However, testing and verification of a system composed of components does not so easily relate to the verification of the individual components. In [109], Leino et al. focus on systems with high level of data abstraction and information hiding, both of which naturally follow from independent development, composability, and coherence characteristics of components. In their paper, they define the property of "modular soundness" as the property that holds when separate verifications of the individual modules of a program suffice to ensure the correctness of the program. They point out that modular soundness is "surprisingly difficult to achieve". Nevertheless, we believe that contracts provide at least the criteria for unit testing, and may be useful in guiding integration tests.

2.3.6 Composite Components

A set of interacting components can be regarded as a composite component. This presents an orthogonal dimension to composition approaches that will be presented in the next section. In this section, we look at different ways of forming composite components, and other related issues.

Aggregation

Aggregation is a simple clustering of component instances, where some of the interfaces of the aggregated instances are exposed to the outside of the aggregation in order to appear as the aggregation's interfaces.

Containment

In contrast to aggregation, the interfaces of the component instances that take part in a containment style composite component are not available to other component instances that are outside the containment. A containment has its own set of interfaces. In the implementation of the services it provides, a containment may use the contained components, but the contained components are shielded from the outside.

Shared Components

Sharing components between different composites is an idea that is relatively scarce. The only example appears to be the Fractal architecture (see Section C.5). The use of shared components can be viewed as a way to create an intuitive design pattern for representing connections between composite components, where the connections have a considerable amount of associated behavior. A good example is provided by Dalle [43, 44, 45], where he proposes representing the simulation of the shared medium of communication in network simulations by using a shared component in the composite components that represent the nodes in the simulated network.

Representing Composites in Flat Component Models

Composite components do not add to the representative power of a component model. Aggregations, containments, and shared components all have equivalent structures in flat models. However, it must be noted that an analysis of the usefulness of such constructs in making the job of the designer easier, is a subject we do not intend to address here.

Aggregations are very easy to represent as a set of component instances. The component instances in the aggregation already have their interfaces exposed. However, an ambassador component for the aggregation can be constructed for encapsulation purposes. This ambassador would forward the traffic coming from outside the aggregation towards the proper interfaces in the aggregated component instances, and the traffic originating from the aggregated component instances towards the outside. A problem with this approach is that the ambassador component can be perceived as a potential performance problem.

A containment can be represented using one logically outer component instance, which solely controls the references to the logically contained component instances. This outer component for the containment can again be referred to as the ambassador for the containment, following the naming we have introduced for aggregations. The behavior of the containment, which is supposed to be implemented by making use of the services provided by the contained components, can be implemented in the ambassador. Since the behavior of the containment has to be implemented somewhere, the addition of the ambassador component in containments cannot be considered as a source of unnecessary performance penalties.

Finally, shared component instances can be viewed as component instances that break down the encapsulation behind an ambassador component instance in a containment or an aggregation, by being directly connected to component instances that take part in different composite components.

2.3.7 Composition

Szyperski identifies two methods for composing component instances into systems: context-based and by wiring [173]. A component model is considered to be using context-based composition if component instances are composed into the rest of the system by placing them in containers that provide the execution environment and services the instances need. In this way, context-based composition emphasizes the role of frameworks as the glue between the component instances.

In composition by wiring, either interfaces of component instances are connected to each other, or component instances are provided with the information about which interfaces of which other component instances they are going to use.

Hybrid approaches are also possible, such as by wiring containers to other containers. The Fractal architecture (see Section C.5) can be considered an example to this approach.

2.3.8 Component Reuse

Reuse is one of the main goals for the component-based approach to software engineering. However, contrary to what appears to have become an urban legend, having mere components is not enough for effective reuse in and across different software systems. In [151], Ran discusses that reusing non-trivial software components is not an easily achievable goal, and argues that component reuse is only possible if it is a consequence of architecture reuse. Similarly, in the context of definitions provided in Section 2.3.3, we will hypothesize that reuse of a component is possible only

- across two component-based frameworks that provide compatible contexts for the component to be reused,
- across two component-based frameworks that are based on the same component model and supported by compatible component platforms,
- and when the component-based architectures related to the two frameworks share compatible associations between entities in a domain and the component model.

Type Systems

Type systems play an important role in expressing where a component can be reused. Three type systems can be identified in component-based approaches: for components, for messages, events, or data communicated between components, and for interfaces. These type systems do not necessarily appear as separate systems from each other in component models, but they can be separated. Identification of uses or usefulness of different constructs in type systems for these three different classes of constructs, such as support for polymorphism or inheritance, will not be addressed in this thesis. Interface definition languages, such as CORBA IDL, address the need for type systems for communicated data and interfaces. Inheritance in components or interfaces is also addressed in various different component models.

Versioning

Since components are independently developed and tested, their evolutions as software products proceed independently. Therefore, versioning and effective use of versioning in reuse of components also appear as problems worth investigating. Versioning can be applied not only for the components, but also for interfaces, contracts, and even for messages exchanged between component instances.

Complexity of the Component Selection Problem

Composing components is a tedious task, as evidenced by the general feeling in the software engineering community. Therefore it is desirable to provide tools for assisting or preferably automatizing this task. For this purpose, the problem of component selection and its complexity has been the subject of various studies in the recent years, especially in the simulation domain.

The component selection problem is defined as follows: given a set of components and a set of objectives, determine whether a subset of K elements of the component set exists, such that the components in this subset collectively meet all the objectives in the given set of objectives [12]. Results are reported in the literature that even in the case where an oracle exists which can provide the information about whether an objective has been met by any component in a set of components, the solution to this problem is NP-complete[6]. This has led to development of approximation methods both in the simulation domain [61] and in other domains [168].

While the original component selection problem assumes black-box reuse of components, reuse with adaptation has also been addressed. The difference in the problem formulation for the component selection problem with adaptation is that components are allowed to be modified, and the modification costs are also taken into account. Bartholet et al. report that reuse by adaptation also turns out to be NP-complete [12].

2.3.9 Relationship Between Object-Oriented, Component-Based, and Service-Oriented Approaches

The borders between object-oriented, component-based, and service-oriented approaches in software engineering are somewhat thin. Component-based approaches are generally expressed in a way that involves object-orientation, and service-oriented architectures appear as component-based implementations of software systems, where component granularity is coarse. Furthermore, these approaches must not be taken as exclusive approaches, but as complementary ones: object-orientation is useful in implementing components or run-time infrastructures for component-based or service-oriented systems, and services can themselves be implemented using object-oriented or component-based techniques.

2.4 Component-Based Simulation

Any new approach in software engineering creates new hopes for a silver bullet. Component-based software engineering did not fail to create high hopes, either. As it happened with object-orientation, a long list of lucrative benefits have drawn attention from the simulation field. Numerous studies that talk about organizing simulation models and their implementations as components have been published. In this section, we will first have a look at the motivation in using component-based approaches in simulation. Then, we will identify some component concepts used in simulation, which should be distinguished.

2.4.1 Motivations

A characterization of various motivations reported in the literature can be observed from the conceptual map in Figure 2.7. There appear to be three main motivations for using a component-based approach in simulation: composability, reuse, and what we will call the simplicity/intuitiveness/less effort argument.

[6] Bartholet et al. cite Petty et al. for this result. For citation, see [12].

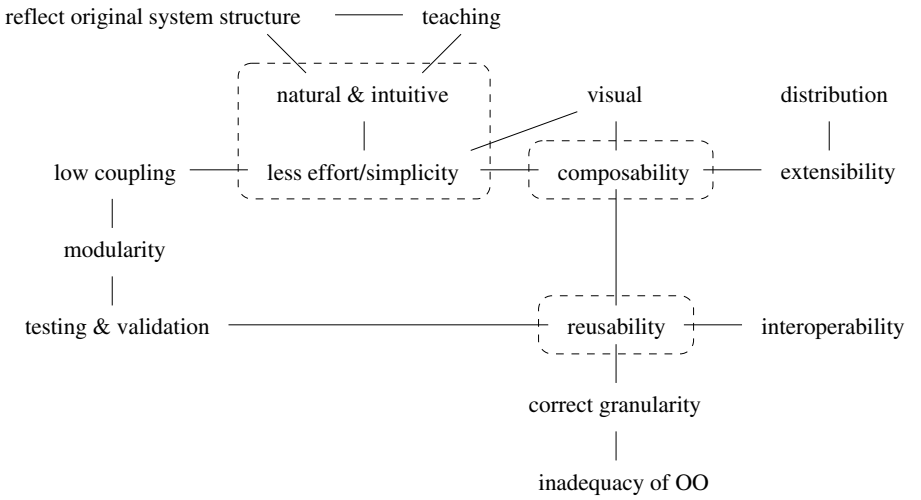

Figure 2.7: Various motivations for using a component-based approach in simulation. The keywords in the dashed boxes appear as the central motivations.

Composability

Constructing simulators or models as compositions of components is a frequently reported motivation [29, 65, 84, 124, 148, 152, 160, 177]. Some authors point out that using visual composition tools for composing components is a promising idea for increasing usability [65, 186]. Furthermore, component-based approaches are expected to be a useful tool for building distributed simulators, or simulators that work in heterogeneous environments [124]. In addition, component-based systems are regarded as more extensible [124, 177], which appears to relate to the independent extensibility property of component-based systems (see Section 2.3.1).

Reuse

Reuse is one of the strongest expectations, and is presented as a motivation for using a component-based approach [13, 29, 37, 38, 43, 45, 54, 65, 107, 145, 166, 170, 177, 183, 186]. Chen et al. make the observation that there appear to be two levels of reuse in simulation: model-level reuse through model replacement, and simulator-level reuse or interoperability [37].

Another theme related to reuse is the inadequacy of object-oriented approaches in satisfying the needs [165, 177]. It is pointed out that binding between objects in object-orientation is too strong [165, 177]. Components are expected to be loosely coupled [29], especially when considered in comparison to objects in object-orientation [45]. In addition, components are perceived to have the right granularity with respect to models in simulations [30], where the granularity of objects is deemed too small [177]. Both Buss and Tyan also refer to components as being generic [29, 178]. Being generic appears to be a concept related to components being at such a granularity level that it becomes easier to identify units that can be reused with adaptation.

As a side effect of reuse, some authors point out potential gains in testing, verification, and validation [45, 145, 152, 170], drawing upon the fact that components are pre-tested units. In fact, Szymanski goes almost as far as saying that the verification problem is solved [170]. How-

ever, potential gains in this aspect of reuse might be limited, due to difficulty in achieving modular soundness (see Section 2.3.5). Carnahan et al. discuss how simulation specific characteristics that are not always found in general software systems might be exploited in order to support composability and reuse in building simulators [32].

Simplicity/intuitiveness/less Effort

Simulation researchers appear to find component-based approaches natural and intuitive [37, 41]. Component-based approaches are considered to reflect the structural organization of the system to be simulated [45], which contributes to their intuitiveness. Being intuitive, they are also expected to be beneficial for teaching different strategies for simulator design [84].

Another motivation that is closely related to intuitiveness refers to component-based approaches as being simpler and as requiring less effort [13, 38, 45, 65, 124, 148, 152, 186]. Component-based approaches are perceived as able to divide systems into manageable smaller tasks [38], and as necessitating less low-level coding [124]. Such a perception is apparently due to the coherence, low-coupling, and independent development characteristics of components. In addition, use of components is expected to lead to less effort being spent in testing and validation [45, 145, 152, 170], and to make it easier to apply "many software-engineering good practices" [45]. Finally, as mentioned before, visual composition is expected to ease simulator construction by turning most of the implementation into composition [65, 186].

2.4.2 Component Concept at Different Levels

Component is a very general concept that relates to the state of being composed. Therefore, components can be observed in different activities involved in the construction of a simulator. Three identifiable uses of components can be distinguished:

Components of the SUT refer to the separately identifiable entities in the SUT or its relevant context in an experiment definition.

Model components refer to coherent, self-contained sub-models of the model of the SUT and its relevant context.

Simulator components refer to the units of software implementation used for structuring the simulator. Simulator components are components in the component-based software engineering sense.

It should be noted that implementation of the model components may or may not be based on simulator components. Even if the model of the SUT and its context is composed of model components and the simulator is built using simulator components, it does not necessarily hold that there exists a one-to-one mapping between the model components and the simulator components.

The distinctions between these different uses of the component concept in simulation are not always adhered to or observed in the literature. Especially in the network simulation domain, emulation-based experimentation blurs the separation between the components of the SUT and the simulator components. Model component and simulator component distinction is becoming more identifiable, through increasing interest in model components level approaches such as DEVS [40,

197], and work on model level libraries [15, 166, 183]. Articles that mention the mapping between model components and simulator components are rare [153].

Four components of component-based approach revisited

Since we have acknowledged that all these concepts are variations of components, one is tempted to explore how they relate to the terminology discussed in Section 2.3.3: component model, component platform, component-based architecture, and component-based framework. These terms were defined for software components, which correspond to simulator components. Therefore they apply without modification for the case of simulator components. Any possibility of using these terms at the level of components of the SUT depends entirely on the domain. To be used at the model components level, their meaning should be stretched a little:

Component model: The definition that was provided in Section 2.3.3 can be used. For example, DEVS formulates components at the modeling level, thereby providing a component model.

Component platform: Sets of algorithms that describe how the components as defined in a component model would work together, can be called a component platform. Again, DEVS can be given as an example: there are different algorithms reported in the literature that describe how components in DEVS can be simulated.

Component-based architecture: An explicit documentation of a particular ontological[7] approach in modeling using a component model, can be called a model-level component-based architecture. It would mainly deal with the mapping between things in the domain being modeled, and the components in the component model being used.

Component-based framework: A model-level set of contracts that stands for a set of reusable design goals. As an example, one may consider the model of a computer where the contracts between different functional units are fixed, but different models of these functional units might be employed.

Granularity and composition of simulator components

Simulator components level composition approaches can be put on a scale with respect to the granularity of the components being composed. In systems such as DIS, HLA, and the Dynamic Simulation Backplane (DSB) [155, 193], a composition has a coarse granularity, since the components being composed are typically whole simulators. In systems with fine grained compositions, the components are typically small models or sub-models, such as individual protocol implementations in network simulators.

Composition styles used in component-based simulation systems or libraries appears to be dependent on the granularity. For systems or libraries that provide fine grained compositions, the main style of composition appear to be composition by wiring [9, 37, 65, 106, 107, 124, 163]. The so-called port-based design [49] can be regarded as just another formulation of composition by wiring. On the other hand, composition style in the systems that use a coarse grained composition resemble more to context-based composition.

[7] In philosophical sense.

2.5 Network Simulation

Network engineering and networking research involve non-trivial tasks such as protocol verification and analysis, and examination of the interactions between various protocols. Using simulators to experiment with virtually created scenarios is an important method that is being used frequently. The perceived importance of simulation has increased in the last decade due to increased interest in understanding large scale networks and building new ones, and due to new developments in wireless network technologies.

In this section, a brief overview of network simulation will be presented. We will divide the presentation of the related issues into three sections on model design and construction, techniques and algorithms for executing models, and engineering of network simulators as software.

2.5.1 Model Design and Construction

Design and construction of models and sets of models is a subject in simulation that relate directly to the domain in which simulation is being applied. Therefore, it is also an integral part of the research method of a networking researcher who uses simulation. It should be noted that from the simulation perspective, model design and construction is also related to meta-models and simulation world views, and various algorithms for executing models.

There appears to be three approaches used in modeling networks with regard to how the traffic is modeled. One approach focuses on representing traffic similar to fluid flow. This approach attempts to come up with a set of differential equations that are solved analytically or by simulation in order to obtain the desired information. In the more popular approach today, all events occurring related to the individual packets are represented in the simulator, leading to a very fine grained traffic model. However, such fine grained models lead to performance problems since they create too many events. In order to harness the benefits from both fluid flow and packet-level models, hybrid modeling approaches that focus on coexistence of flows along with packets, are also being researched [2, 98, 101].

In addition to the work on developing ways of modeling networks, there is also considerable work on developing simulation frameworks, simulators, and model libraries. A simulation framework for a particular domain provides the description of what family of models should be interacting with which others, and in which patterns. That is, a framework is concretized by putting models into the placeholders defined through the framework. In reference to the architectural tiers discussed by Szyperski in [173], three approaches to development of simulation frameworks, simulators, and model libraries can be distinguished:

- Implementing from scratch: x-Sim [22], NS [23], SIMMT-II [31], GENESIM [50], the simulator by the University of Durham and Terrington Systems Ltd. [53], WIP-Sim [102], DISDESNET [125], GTNetS [154], OMNeT++ [182], Nessi [184].

- Implementing as additional architectural tiers on existing general purpose simulators, simulation libraries, or simulation frameworks: SWANS on JiST [10, 11], Libra on Ptolemy [39], MaSSF on SSF [113, 114], IRLSim on Parsec [175], J-Sim on ACA [177, 178], ATM-TN on SimKit and WarpKit [179], GloMoSim on Parsec [198].

- Implementing as additional architectural tiers on existing network simulators: MNS on NS [1], GEO on NS [96], GTSNetS on GTNetS [138], SensorSim on NS [140].

2.5.2 Techniques and Algorithms for Executing Models

Subjects of discussion on the issues related to techniques and algorithms for efficient execution of models compare well to the well known division between time and space requirements of algorithms.

The memory usage becomes a serious problem as the scale of the network to be simulated gets bigger, since all events generated, packets in transit, and executable models including their state information consume the memory [64]. Different approaches target this problem, such as reducing the number of events [2, 154], and partitioning models for distribution onto multiple parallel hosts [113, 132, 171].

Decreasing the running time needed for simulations of network models is another important problem. Some benefits of decreasing the running time are:

- It would allow better utilization of researcher time.
- More experiments can be carried out, which might lead to more insight.
- More episodes can be executed, which would lead to better statistical results.
- Sometimes ability to decrease the running time might be the factor that makes an experiment feasible to conduct, which is the case for simulation of large scale networks.

Work on more efficient event list implementations [139, 192, 194], reducing the number of events [2, 154], and optimizations such as no-copy message passing between models and the scheduler [11] aim to improve the performance of network simulators, regardless of the simulator being parallel, distributed, or monolithic.

Using parallel and distributed simulation [63, 125] is another popular way of attacking the problem of decreasing running time. There exists both parallelization of already existing network simulators such as NS [64, 111], and network simulators designed from the outset to be able to run in a distributed manner [154]. However, as Perumalla et al.'s work shows, there is still work that needs to be done in implementing these techniques in the run-time infrastructures (RTI), as even small optimizations may decrease the RTI cost in event processing from 55% to 18% [143]. Whether and under which conditions parallelization of a model has a sufficiently high utility from the point of view of the experimenter [127] is also an important factor in choosing simulators from the experimenter's point of view. This is also important from the network simulator developers' point of view for identifying strategical targets, in terms of model size and distribution options provided in the simulator.

2.5.3 Engineering of Network Simulators

On the engineering of network simulators' software, there is not much discussion in the literature. Usability of network simulators appears to be regarded as solved by providing visual interfaces or a scripting interface [23] for configuration and scenario creation. Extensibility is generally associated

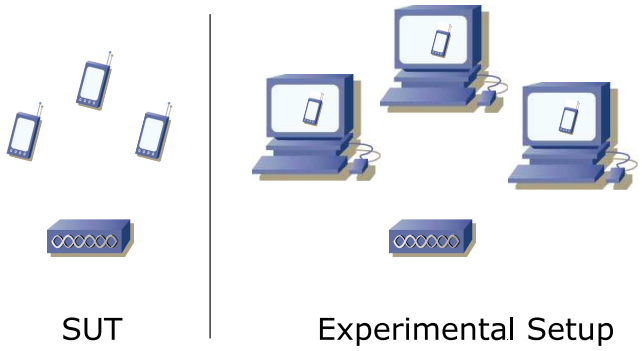

Figure 2.8: Example to using physical layer of the SIFSUT as a surrogate for the physical layer of the SUT.

with being an object-oriented design or a modular-design. The architecture of network simulators is perceived as being a layered one, where a simulation kernel forms the lower layer, and model implementations form the upper one.

2.6 Network Emulation

We have published a survey on network emulators and testbeds in [73]. That survey has been included in this thesis as Appendix B, with small revisions. Based on this survey, this section provides an overview of techniques used in network emulators.

In some of the systems reported in the literature, identifying what is simulated is not trivial. For these systems, what is simulated depends mostly on the definition of the SUT and its context in the experiment [36, 144].

It is difficult to model the physical layer in networks accurately, and simulation of the physical layer is computationally intensive. Using the real physical layer of the experimental setup as a surrogate for the physical layer of the SUT [36, 108, 144, 181] is one of the approaches used to address this problem in the case of wireless networks (see Figure 2.8). In this approach, the wireless medium in the experimental setup is either used directly [36, 144], or by optionally adding noise [108, 181]. When using the wireless medium of the experimental setup, the distances between the nodes in the SUT or its context can be scaled down by attaching signal attenuators on the antennas of the nodes used as real in the experimental setup [47, 95, 181]. As an alternative to using the wireless medium in the experimental setup, some researchers have explored capturing and feeding the radio signal by directly interfacing with the antennas of the wireless NICs of the nodes that are used as real. The captured signals are processed or guided to create desired effects and connectivity [93, 94, 95].

When the physical layer in the experimental setup is used as a surrogate for the physical layer of the SUT and its context, there may be a need to address mobility as well depending on how the SUT and its context is defined in the experiment. Methods used for simulating the mobility of the nodes include using cars [120], using people to carry nodes around [115, 130], and mounting the nodes on small mobile robots [47, 181].

Routing traffic from and to the hosts that are used as real in the experimental setup through

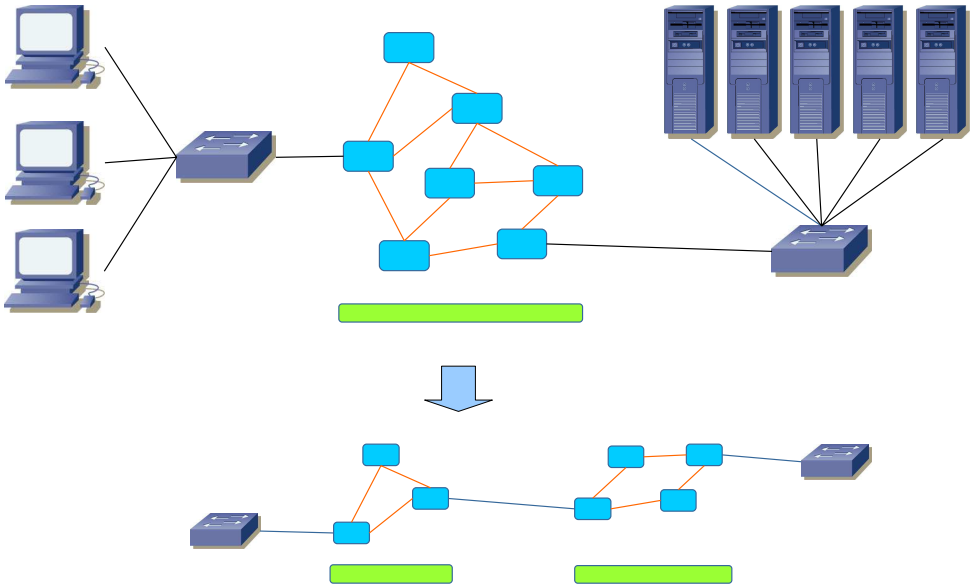

Figure 2.9: Example to emulation by routing traffic from real hosts through a simulated network. It is also possible in this approach to use multiple simulators running on different hosts, which simulate different parts of the simulated network, as shown in the bottom part of the figure.

a simulated network is another emulation technique [57, 78, 100, 112, 118, 161, 162, 180] (see Figure 2.9). In these studies, modification on the hosts used as real is typically very limited, such as configuring them to route their packets through the simulated network. A more intrusive alternative is placing traffic shapers between protocol stacks and network interfaces in the hosts providing the traffic [81, 82, 117, 200, 201] (see Figure 2.10). While these traffic shapers may or may not be controlled by a central server, this approach can be considered as distributing some of the functionalities of the simulator to the hosts that are used "almost" as real.

An alternative to using traffic shapers in the kernel is using the universal TUN/TAP driver, which provides virtual network interfaces. Using these interfaces, it becomes possible to use the protocol stack implementation in the kernel and the applications working on top of them as real (see Figure 2.11). Furthermore, a simulator running in the user-level can simulate the physical layer and lower layer protocols [74, 99, 149, 185].

Another approach mainly focuses on using the protocol implementations in kernels as real. Some studies pack protocol implementations from operating system kernels in a way that is usable in the user-level [55, 86] on top of the kernel, or as a model in a simulator [19, 91]. Having multiple copies of the protocol stack in a single kernel is also possible [196]. Similarly, there also exist studies that uses multiple kernels or operating systems working concurrently on a base operating system [56, 77, 92]. These approaches are graphically shown in Figure 2.12.

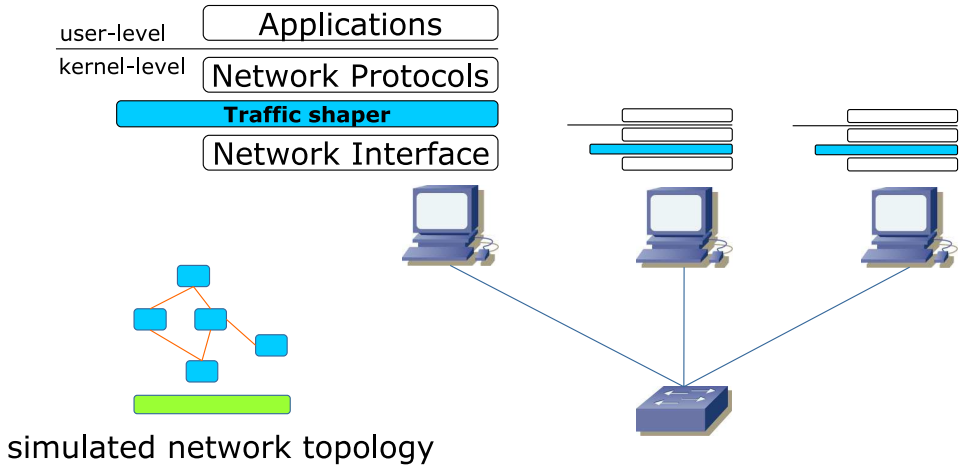

Figure 2.10: Example to emulation by using traffic shapers.

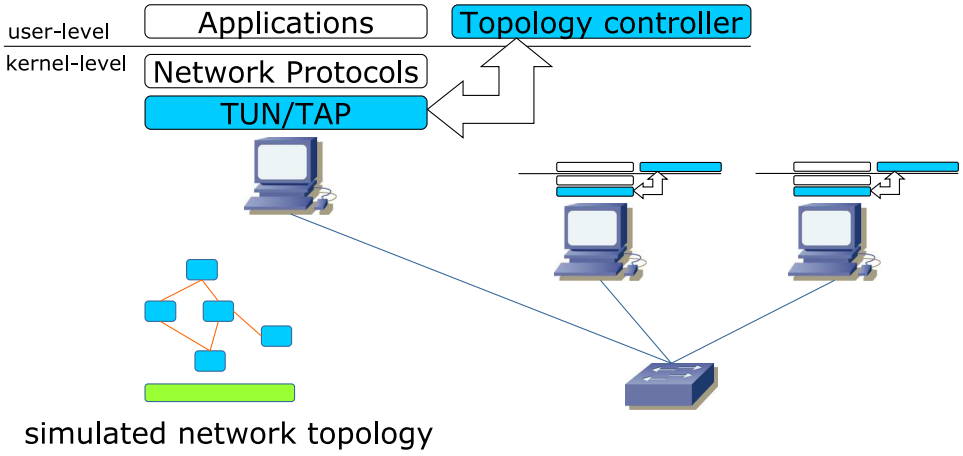

Figure 2.11: Example to emulation by using TUN/TAP virtual network interfaces.

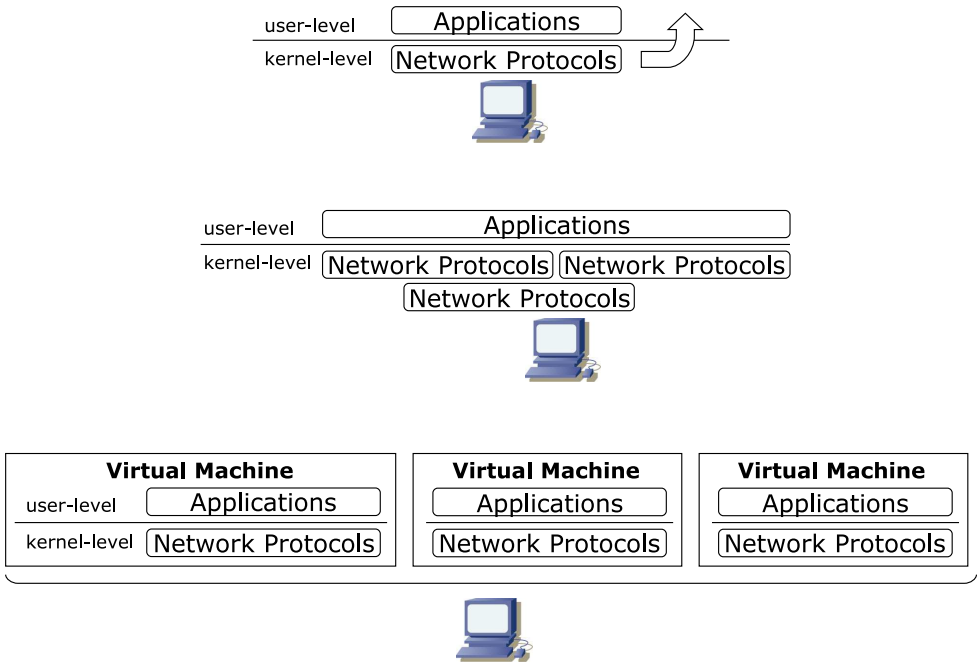

Figure 2.12: Different approaches in using protocol implementations as real in network emulators.

2.7 Component-Based Network Simulation and Emulation

Component-based approaches to simulation were previously discussed in Section 2.4. The three different uses of the concept of components in the context of simulation presented in that section can also be observed in the studies on network simulation and emulation.

In fact, networks are perceived as composed of network components, such as nodes, routers, switches, links, and so on. The components of the SUT are readily identifiable to the network researcher. This leads to the observation that composition of the model to be simulated from models of the individual components of the SUT, would lead to an intuitive method for building simulators or emulators for networks. Therefore, some of the network simulators that claim to be component-based, such as those reported in [54], [75], [146], or [169]

- regard the component-based nature of the networks as their component model without further elaboration,
- derive a component-based architecture for the target networks at the model level,
- implement a framework, which is not necessarily component-based, for the specific architecture they have developed.

Since the component model is usually not explicit in these systems and no distinction is made between the framework and the simulator, there usually does not exist a separate component platform.

In the rest of this section, four approaches to building network simulators will be shortly discussed: TeD, OMNeT++, OSA, and ACA. These four are chosen based on their relevance to this thesis.

2.7.1 TeD

Bhatt et al. present the TeD as an object-oriented framework for implementing network simulators. Their motivation is that the communicating logical processes that are managed by a parallel simulation kernel are fairly low-level constructs for network researchers [18]. While introduced as an object-oriented framework, TeD comes closer to being a component-based approach more than many other network simulators which claim to be component-based. Bhatt et al. write that the entities in TeD are to correspond to models of "the physical and conceptual objects in the telecommunication domain". Their description therefore points towards what have been named as "model components" in Section 2.4. An entity in TeD contains four architectural constructs:

Deferred constants: Variables whose values can be set at configuration, but they remain constant during a run.

State variables: Variables whose values form the state of the entity, which may change during a run.

Processes: Behavior descriptions which are associated with event arrival, or time advances.

Components: Entities encapsulated by the outer entity. These so called components are used for implementing the processes of the outer entity. In this sense, TeD supports containment style composite components (see Section 2.3.6).

The specification of the control flow in TeD focuses on processes. Each process is associated with its own computational context, typically a thread. The advancement of the logical time is established using different forms of `wait` statements in the implementation of a process.

At the implementation level, a TeD entity is represented by its description in the TeD language. Processes are written using a general purpose programming language (C++), and they are referred to from the TeD description of the entity. The descriptions of the entities to be used in a simulator are compiled using the TeD compiler, along with their process descriptions and a parallel simulation library, in order to create a simulator. The resulting simulator is configured using a configuration file, which describes the run-time composition of entities, and their initial configuration.

We said in this section that TeD comes close to being component-based. TeD certainly describes a component model. It appears from the motivation presented for the entity abstraction that one of the goals in TeD is to harness the benefits of alignment between components of the SUT and model components (see Section 2.4). However, the argument they present in favor of layered architectures for network simulators indicates that instead of aligning the model components with simulator components as well, they aim to shield the network researcher from the simulator implementation details by abstracting them behind a compilation phase. Therefore the component platform does not exist as a separate software in TeD, but it is compiled into every simulator.

2.7.2 OMNeT++

OMNeT++ is an open-source network simulation package designed by András Varga [182]. OMNeT++'s design goals were influenced by his position on model building for network simulators: "To enable large-scale simulation, simulation models need to be hierarchical, and should be built from reusable components as much as possible".

OMNeT++ describes a component model that is mainly targeted for model components. The components in OMNeT++ are called modules, which are expected to be independently developed by different researchers or developers.

The modules in OMNeT++ can communicate using either their interfaces, which are called gates, or directly by calling methods of objects in their implementations. This second way of communicating has the potential to introduce many direct dependencies, and can be considered as an object-oriented trait. The gates are further divided into in and out gates, according to whether data flows into or out of the module.

The composition style in OMNeT++ is composition by wiring. A specially designed configuration language called NED is used for describing which components are to be instantiated, and how the gates of the instantiated components are to be wired together. OMNeT++ modules can be combined to form composite components in an aggregation style. The composite module thus created exposes a select set of gates of its aggregate modules as if they are its own gates. Non-composite modules are called basic modules. Containment style composites are not directly supported, since composite modules do not have behaviors of their own which can use the contained modules.

The modules in OMNeT++ are not packed as self-contained binary units. They exist as a set of C++ files, which are compiled into a simulator along with the simulation management functionality (SMF, see Section 2.2.1). The SMF is provided as a class library, orthogonal to the component model, similar to the case for TeD. There appear to be APIs available for implementing both monolithic and distributed simulators.

There are two execution models supported by OMNeT++ for the basic modules: co-routine based, and run-to-completion on message reception. The co-routine based model is recognized by Varga to consume more memory, since it requires one stack per module.

It appears that OMNeT++ either does not aim to be fully component-based at the model or simulator components conceptual levels, or it fails to be so. The association of link characteristics between two communicating entities in the network being modeled with properties attributed to the connections set up between modules, breaks down a clean alignment between the components of the SUT and model components. In this way, behavior is associated directly with the wiring between the components, introducing implicit components into the component model. In addition, allowing direct calls between the modules increases the coupling and weakens the module coherence, making it harder to recognize them as components.

Despite all its weaknesses, OMNeT++ is a good attempt in aligning all three uses of the concept of components in building network simulators. Furthermore, it stands as an example to the fact that such alignment is indeed considered to provide an intuitive tool for the network experimenters who need to build network simulators.

2.7.3 Open Simulation Architecture (OSA)

The Open Simulation Architecture (OSA) [43, 44, 45] is a component-based approach to discrete event simulation using the Fractal component model (see Appendix C.5). While OSA is not only for building network simulators, it will be included in this section since it is developed in relation with a project targeting the networking domain.

The OSA adds onto the component model how the simulation management functionality (SMF) will be served to the components in the simulator. OSA suggests using a shared object hidden in the membrane of components for implementing the SMF, and present the component with an interface in its membrane. This object can also be made into a shared component since Fractal allows componentization of component membranes. Another option is making this object into a component that is shared between different composite components that act as model implementations, but this is considered to lead to a design that mixes functional and non-functional components.

As discussed in Section 2.3.6, the use of shared components in modeling and implementation is an interesting subject. While it may turn out to be beneficial, further studies seems necessary for understanding the potential benefits. An initial step in this direction can be found in [46].

2.7.4 Autonomous Component Architecture (ACA)

The motivation behind the Autonomous Component Architecture (ACA) [177, 178] is the classical idea of building software using the integrated circuits (IC) as a metaphor, inspired by the success of ICs in the electronics industry. ACA is used as the underlying architecture for the simulator J-Sim.

Components in ACA communicate by exchanging messages through their ports. These ports are simple outgoing and incoming points for data. Components are composed by wiring their incoming and outgoing ports. Such wiring is assumed to be done in a way that satisfies the contracts between the components. The contract between two components whose ports are wired together is expected to define the content of the data exchanged through their ports and what is to be expected from the components when data is sent or received. While these assumptions are made, contracts are not defined or managed by ACA.

In addition to wiring components together, it is also possible to compose components to form composite components in ACA.

ACA component model leaves considerable amount of design decisions to the component platform implementer. How naming of ports, and creation and deletion of components and connections are to be done, are all left to be determined by the component platforms for ACA. This have lead to the fact that there is a single component platform implementation for ACA, to the best of the author's knowledge, and it is the one that is started by the researcher who have also formulated the component model [177].

An interesting feature of ACA is how it handles the control flow. The main model of control flow is called the independent execution model. In this model, when a component receives data from a port, it immediately processes it in a new independent execution context, which in practice means a process or more likely a thread. There is also support for a blocking send operation, where the sending component is blocked until the receiving component finishes processing the data that was sent.

Chapter 3

Elaborating on Network Simulation and Emulation

In Sections 2.4, 2.5, and 2.6, we have surveyed the tools, approaches, and techniques used in network simulation and emulation. In this chapter, we further elaborate on network simulation and emulation by presenting our observations. We start by having a look at where simulation and emulation fits in developing networks and network applications. Then we present two roles involved in simulation and emulation experiments, and argue that these roles have conflicting goals. In Section 3.3, we have a closer at the process of one of the roles. We close the chapter by presenting different approaches to integrating real entities with emulators, followed by the problems specific to emulation that do not show up in simulators. An early version of Sections 3.1, 3.4, and 3.5 were published in [69].

3.1 Simulation and Emulation in Development

The role of emulation-based experimentation for communicating systems, is one of bridging the analytical or model level development stage with the actual real-life deployment of the system, in order to help the transition between these two levels to happen in a smoother way. Although a very useful tool for conceptual analysis, pure simulation studies might not always provide consistent and relevant prediction or verification [35], or might miss important effects [130]. Therefore the whole spectrum from pure simulation experiments up to real system test deployments seems necessary for developing robust systems, and specifically for systematically managing cross-layer concerns in MANETs.

Emulation-based experiments are not a replacement for simulations or controlled test system deployments [62]. The emulation integrated development (see Figure 3.1) may start with the simulation stage when the algorithms are available while the actual implementations or the target platforms are not. Enrichment of the simulation environment is advised to start as soon as implementations of various components, of possibly limited functionality, become available [82]. The introduction of first implementations of the components of the SUT (see Section 2.4.2) into the simulations is the point where the emulation-based experiments start. Emulation-based experiments incorporate the whole spectrum from this point up to controlled test deployments of the system.

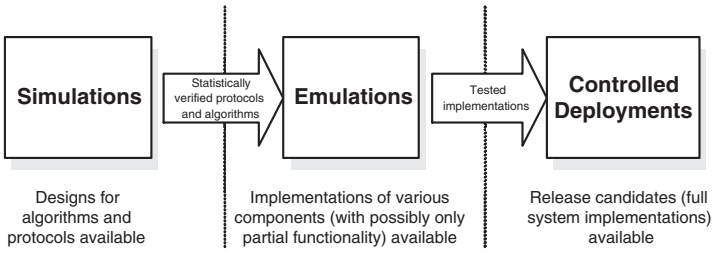

Figure 3.1: The emulation integrated development.

3.2 Parties Involved in Network Simulation and Emulation

Two different parties are involved in realizing simulation or emulation-based experiments about networks: the experimenters, and the simulation system or library developers. Although these two parties have different interests and goals, and follow different steps of actions before and during an experiment, researchers typically assume both roles without paying much attention to the differences.

The commercial systems provide a much clearer separation between these two stakeholders, since they depend on this double stakeholder structure to create a network simulation systems and libraries market. The downside is that academic studies is hindered because the code and the algorithms are usually not disclosed, or are protected under copyright.

We discuss in this section that the differences between these roles are significant, and the goals of different parties seem to make it hard for one party to act in the other role as well.

3.2.1 Experimenters

The experimenters are the researchers who are interested in conducting experiments whose goal is to understand the behavior of networks under particular situations and workloads.

An experimenter has two major goals with respect to the simulation or emulation-based experiment. The first is the goal of *minimum effort*, which can be described as the wish to design, implement, and run a suitable experiment with minimum resources, especially time. The other goal is to build and realize the experiment in a manner that will produce meaningful data for the analyses planned to be carried out. This second goal can be referred to as the goal of *adequately meaningful data*.

We will have a look at the experimenter's process in Section 3.3.

3.2.2 System and Library Developers

The other party to simulation or emulation-based experimentation about networks, is the developers of the simulation systems and libraries. These developers have the following as their goals:

- Provide a straightforward and easy to follow mapping between the concepts in the domain, the elements of the metamodel used for modeling the target systems in the domain, and the elements of the software architecture used for implementing the run-time representation of the model in the simulator to be built.

- Provide a simulator library (see Section 2.2.1) with implementations of basic models of conventional network hardware and software. Such a library is provided to the experimenters to use when building the model of the SUT and its context in their experiments. This allows researchers to minimize their efforts to only implementing a limited set of models that is directly related to their research.
- Design the software specifications of the simulation library or system for good performance, and optimize the implementation and the interactions of basic models that are provided in the simulator library.
- Keep it simple but adequate, and document the system well.

The system and library developers focus on the tasks described below for attaining these goals:

- Prepare base software specifications for the simulators to be built using the system or library, in such a way that satisfies the first goal above.
- Decide on how the specifications can be mapped to programming constructs. This mapping should preferably conserve the architectural organization of the possible systems of interest in the domain, and reflect the style of the metamodel used in modeling.
- Design to provide clean interfaces to be used by experimenters when building and composing submodels.
- Implement with clean and understandable code, and provide enough comments. It is preferable to use an open-source approach, since the code is regarded as the most precise description of what the system does. Therefore, disclosing the source code supports the fourth goal above.

3.3 Experimenter's Process

In this section, we take a closer look at the tasks addressed by the experimenters in simulation and emulation-based network experimentation. We will divide the process of the experimenter into four phases as shown in Figure 3.2: preparation of the experiment description, preparation of the experimental setup, executing the experiment, and post-execution analyses.

3.3.1 Preparation of the Experiment Description

The first step in any experiment is preparation of a description of what the experiment is about. The experiment description should include the following:

Specification of the system under test (SUT):
The specification of the SUT involves drawing the boundaries of the system about which conclusions will be drawn from the experiment. While it is a step that sometimes is not deemed much important, or considered too "obvious" to spend time on, inaccurate or imprecise specification of the SUT may lead to inapplicable results and conclusions drawn from the data gathered in the experiment. This specification provides the description of what is to be modeled for the purposes of the experiment.

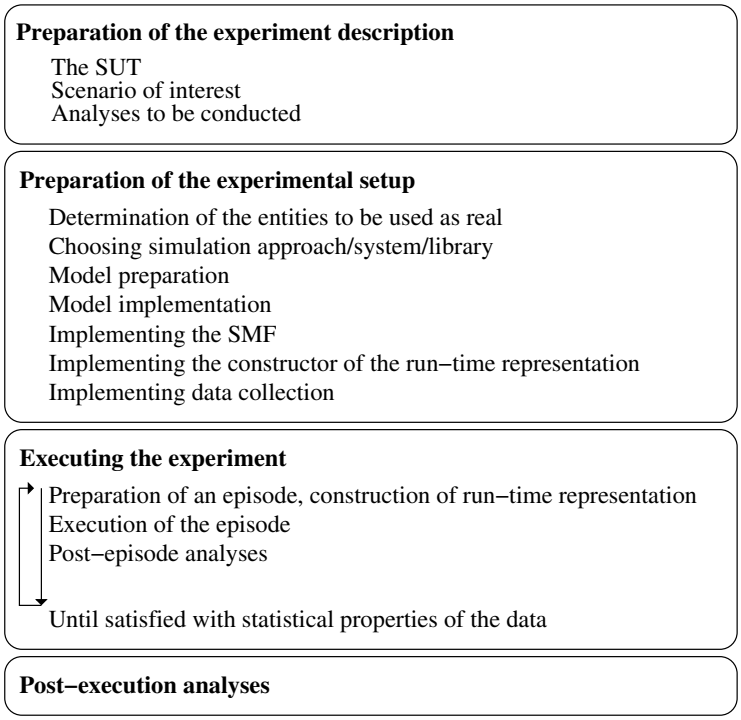

Figure 3.2: Summary of the experimenter's process for simulation or emulation based network experimentation.

Scenario of interest:

A scenario is a description of a particular situation of interest involving the SUT. The scenario should include the initial conditions of the SUT, and the description of the workload the SUT is subjected to in the situation of interest.

Analyses to be conducted:

Depending on the goals set forth for the experiment, analyses to be conducted should be determined and documented. In addition, the statistical requirements or goals for the analyses should be decided, and parameter and factor sets should be set up. Then, what data is needed to be collected to conduct the analyses in accordance with the statistical requirements can and should be determined.

3.3.2 Preparation of the Experimental Setup

Following the description of the experiment, the experimental setup can be prepared. This step involves building the stand-in for the system under test (SIFSUT) to be run in episodes for collecting data. The following should be addressed in preparation of the experimental setup:

Determination of entities to be used as real:

A decision should be made whether a pure simulation, or an emulation-based experiment

will be conducted. If an emulation-based experiment is to be conducted, the entities in the SUT or its context that will be used as real in the experimental setup should be decided. Furthermore, how these real entities will be connected to the emulation capable simulator should be addressed.

Choosing simulation approach/system/library:
The choice of simulation approach, and simulation system or library (see Section 2.2.1) generally turns out to be an intertwined decision because simulation systems and libraries are generally built for supporting a single approach. Furthermore, this choice also partly or fully determine the metamodel that can be used for constructing the model for the simulated parts of the experimental setup.

Preparing the model:
Once a metamodel to be used for constructing the model is determined, either by explicit choice or by imposition by the choice of the simulation system or library, the model of the simulated parts of the experimental setup should be prepared. In construction of the model, the parameter and factor sets, and the data to be collected as specified in the experiment description, should be taken into account.

Implementing the model:
The model of the necessary parts of the SUT should be built. This model may or may not be composed of submodels. If the model will be composed from submodels, and if the chosen simulation system or library comes with a simulator library (see Section 2.2.1), then it might be possible that the whole model can be constructed by composing the models from this library. However, it is usually the case that some models of interest are missing from the simulator library, which then need to be implemented by experimenters in accordance with the software specifications of the chosen simulation system or library. It should also be noted that in addition to the SIFSUT, models that will exert workloads on the SIFSUT also need to be implemented. Furthermore, initial conditions described in the scenario in the experiment description should be mapped to initial conditions of the model implementations.

Implementing the simulation management functionality:
If a simulation library has been chosen instead of a simulation system, the simulation management functionality (SMF, see Section 2.2.1) should be built and configured. In that case, it will be necessary to make additional decisions such as what kind of a scheduler is to be used, or how the event propagation or routing mechanisms will be implemented.

Implementing the constructor of the run-time representation:
A mechanism, usually depending on the simulation system or library, for constructing, initializing, and running the run-time representation of the model and the SMF should be implemented. This takes different forms in different systems, such as a small program, a script, or a configuration file. For example, NS provides scripting language interfaces to describe how the run-time representation will be built, based on the hypothesis that scripting languages are easier to understand and use. A script and the script interpreter in NS acts as the constructor of the run-time representation as we define here.

Implementing data collection techniques:

The data collection is one of the important issues in the implementation of the experimental setup. Therefore how the data will be collected and where it will be logged should be carefully designed.

3.3.3 Executing the Experiment

By execution of the experiment, we refer to running the prepared experimental setup in episodes. The sets of factors for the parameters might be different in different episodes. If there are random variables used in the model, or if there are effects caused by real entities that can be regarded as outcomes of a random variable, then episodes that do not differ in factors might also be run just to obtain bounds on statistical results.

The activities in each episode are the following:

Preparing for the Episode:

In the first phase of the episode, the simulator or simulators for the simulated parts of the experimental setup should be loaded, the real entities should be initialized, and the connection between the real entities and the simulator or simulators should be set up.

Like almost all software, a simulator's stored form and run-time representation are different. Particularly, this implies that the model to be simulated exists in stored form as some number of submodels and a description of how instances of these submodels are composed at run-time to arrive at the actual executable model. Therefore before anything, a run-time representation of the SMF and the executable model should be constructed using such stored forms.

Executing the Episode:

When the run-time representation is set up, the executable model is run, and events related to the analyses to be carried out are monitored and/or logged. The analyses that are carried out during the execution are usually computationally light. These lightweight analyses might serve to summarize the data for efficient logging. Another important motivation is to monitor the experiment for interrupting the episode in a timely manner, in case there is a problem in the scenario or how the scenario unfolds.

A position we take in this thesis is that the the run-time representation of the model should not change during an episode. This relates to parts of the implementation of the simulator or simulators that are being used, call them modules or components, and how they are connected to each other. Changes in how these parts are interconnected, or creating/deleting new parts or simulators introduce problems, such as lost events. Furthermore, we doubt that such changes in the run-time structure of the simulators add any expressive power, or extend the range of models that can be simulated. Any "appearing" or "disappearing" entity in the SUT should be a part of the SIFSUT from the beginning of the episode, to be enabled or disabled at a later stage.

Post-Episode Analyses:

The episode ends when some predefined event happens, which is usually a limit on the virtual

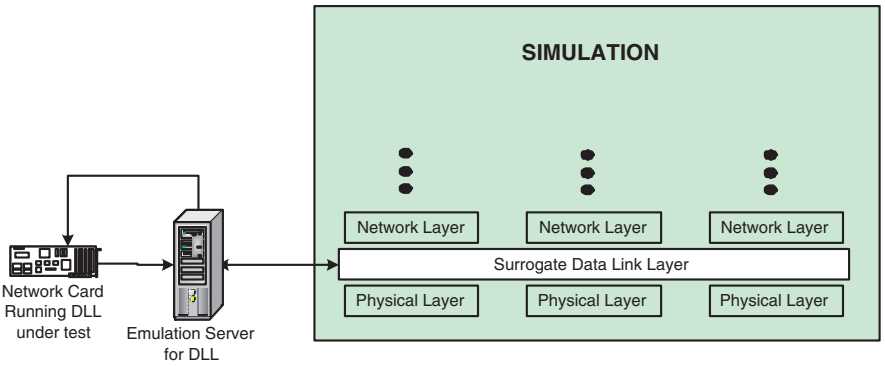

Figure 3.3: Example to horizontal integration method.

time. Then the real entities should be disconnected from the simulators and the simulators should be stopped.

At the end of the episode, additional analyses can be done. These post-episode analyses would be the ones that require more resources than those that are feasible to conduct during the episode. However, the goal of the post-episode analysis is not deriving the final conclusions for the experiment, or formulating any results. Instead, the goals are the following:

- Finding out whether the experiment is progressing as planned. Otherwise, there might be a need to go back and modify the experiment description or setup, which most probably would render the data collected up to that point unusable.
- Summarizing the data collected in the episode, and integrating these summaries into the cross-episode data sets.

3.3.4 Post-Execution Analyses

When all planned episodes are executed, or enough episodes are deemed executed with respect to the goals set out in the experiment, resources can be used for more thorough analyses on the data collected during the episodes.

3.4 Methods for Integrating Real Entities

Real entities from the SUT can be integrated with a simulator in various different ways. Two main classes of integration methods can be identified: horizontal integration, and vertical integration.

In *horizontal integration*, a model for a specific layer functionality is replaced by the actual implementation for that layer in a simulated network architecture (see Figure 3.3). This method is more suitable when the entities from the SUT used as real can't be duplicated in an efficient way, or when there exists a shared resource, like a scarce target platform, which needs to be employed in a time-sharing manner.

On the other hand, in *vertical integration*, the software used as real covers groups of consecutive layers of the network architecture that are organized in a per node basis (see Figure 3.4). Typically,

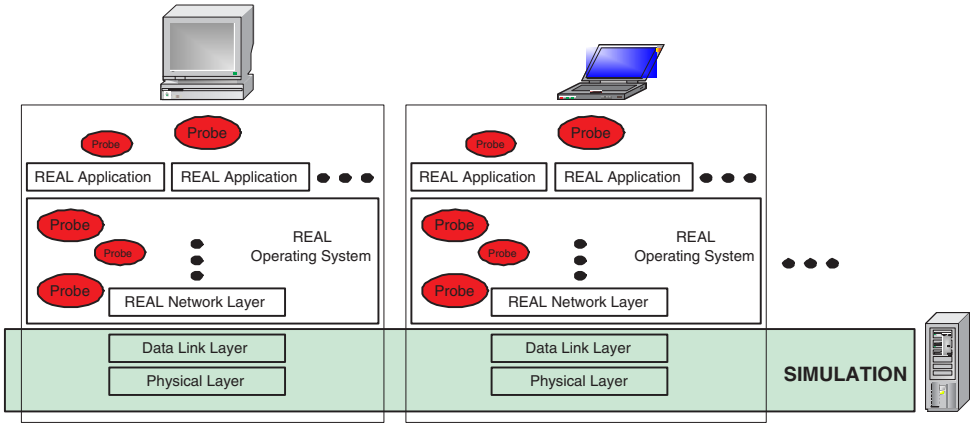

Figure 3.4: Example to vertical integration method. This architecture appears to be most typically associated with network emulation experiments.

data-link layer or IP layer and upwards are used as real, whereas the lower layers are realized by a simulator. This approach leads to an intuitive decomposition and distribution.

These two methods can be and usually are mixed, as exemplified by the some of the TUN/TAP virtual network interface based emulators. In these systems, physical and data link layers are simulated, some of the network protocol implementations in a single kernel are shared, and various real applications work on top of the kernel. This presents an architecture of vertical emulation on top of horizontal emulation, which works on top of simulated lower layers.

3.5 Emulation Specific Problems

In [69], we have presented a list of major potential problems in emulation-based experiments for networks. In this section, we summarize and elaborate on the material that was previously presented in that paper. It should be noted that we have mainly focused on software entities that are used as real in the experimental setup in emulation-based experiments. Furthermore, the reader might want to review how an emulator is defined for the purposes of this thesis, which was presented in Section 2.2.3.

Some or all of data generated in an emulation-based experiment is causally dependent on the simulated models. Therefore, although not included in this section, the problems about network simulators also show up in emulation-based experiments.

3.5.1 Simulation-Emulation Boundary Problems

Running simulations along with real entities from the SUT or its context, creates potential for problems stemming from the simulation-emulation boundary. The *simulation-emulation boundary* can be defined as the total of all data and control transactions between the simulated models and the real entities in an emulation-based experiment. For example, Ethernet frames in a vertical network emulation in which the physical layer is simulated, are among the data transactions that crosses the simulation-emulation boundary.

One problem related to such data transactions is the lower precision and accuracy of the simulated models, which may lead to omission of or inaccuracies in some data that would normally traverse the boundary in the SUT. The impact of such data on the behavior of the entities used as real in the experimental setup should be managed in emulation-based experiments.

3.5.2 Problem of Physical Hosts in Synthetic Environment

In simulations, the models running in the emulator are designed to trace the states of an entity in a virtual (a.k.a. synthetic) environment described in a scenario. The virtual environment itself is composed out of models. Therefore the whole system is one that traces the steps of a totally virtual world, in which space and time are not bound by physical world nor its laws. This in effect means that the physical world, in which the state tracing is done, has no effect whatsoever on the behavior of the entities in the virtual world whose states are traced by the models.

In the case of emulation-based networking experiments, real software entities frequently show up as protocol implementations used as real. The performance and behavior of such real implementations are considerably affected by the availability of computational, storage, and other resources on the physical host that they are running on. Therefore, while the SIFSUT is totally virtualized in the case of simulations, it includes the physical hosts and the resources available on them as non-virtualized parts in the case of emulation-based experiments. For this reason, without proper management and virtualization of the physical hosts on which the real entities are executed, the emulation-based experiments can not be regarded as providing fully synthetic environments in which to test the SUT.

The main problem that comes with inclusion of physical hosts in the synthetic environment has shown itself in the literature as the imposition of real-time constraints over the simulated models in emulation experiments[56, 82, 93, 141]. However, especially when simulating large broadcast mediums with complex physical characteristics, such as in mobile ad-hoc networks, the acuteness of the trade-off between the accuracy and fidelity of the models and the computational resources necessary for simulating them raises concerns[188]. The constraint of matching virtual time to real time, creates a lack of an adequate amount of computational resources available for simulating models. The remedy is sought in simplifying the models [126], thereby giving up on accuracy in order to decrease the computational load of the model to make it conform to real-time constraints. The justifications provided for such simplifications of the models, if any justification is provided at all, usually draw upon the increased scalability as the motivation. However, these justifications generally lack analytical or statistical analyses backing them. Very few papers even mention the problem[82, 141]. An alternative solution for preventing imposition of real-time constraints on the simulated models is the virtualization of the physical hosts themselves. The virtual machine, user mode kernel, and micro-kernel based approaches can be interpreted as the first hints towards this solution (see Section 2.6).

Yet another problem is related to the management of emulation-based experiments. For simulation experiment episodes, it is easy to provide managemental functionality such as pausing, accounting for lag in one or more of the models, saving state, and restoring, because the simulation control system has complete control over the virtual environment. As the current emulation control systems lack complete control on the physical hosts as parts of the experimental setup, such managemental functionality they provide is very limited in comparison to simulation control systems.

This makes the emulation-based experimentation process relatively more brittle, thereby causing more stress on the experimenter. The emulation control systems provide little help if anything that was not accounted for from the outset happens during the execution phase of the experiment.

3.5.3 Problem of Transparency

When the real entities in an emulation-based experiment are software, we can define a scale out of the amount of changes necessary to run these real implementations along with the emulator. We will call such a scale the *transparency of the emulator*.

The transparency of an emulator should preferably be as high as possible, which is the same as saying that changes in the software used as real should be kept to a minimum, and preferably be none at all. Meanwhile, an application or a protocol should run alongside with the emulator as if it were running on its target platform. To accomplish transparency while keeping the results accurate, an emulator should account for many differences between the target platform and the physical hosts on which the real software entities are executed. These differences stem from the amount of resources available, resource contention and conflict characteristics, and different implementations of abstractions at different levels, in particular in the operating system level abstractions. Providing high transparency presents a considerable challenge in the design of emulators.

3.5.4 Emulation Overhead and Emulated Entity Multiplexing

Emulation-based experiments are actually performance analysis studies. In performance analysis, data collection causes a well known problem called monitoring overhead problem [90]. The monitoring overhead problem is the result of using hardware and software probes which themselves have effects on the system performance.

In simulation experiments, the monitoring overhead is not considered as an issue, since the environment in which the models are simulated is assumed to be fully synthetic, as discussed in Section 3.5.2. In a properly designed simulator, the monitoring probes are outside of the synthetic environment and the conceptual model. Therefore, they can not cause any unwanted effects in the events happening in the synthetic environment. However, in an emulation setting, the inclusion of real entities brings the computational platforms they run on into the experiment environment, again as discussed in Section 3.5.2. Therefore, the physical hosts become the gateway to monitoring the status and behavior of the real entities, in effect bringing monitoring overhead effects into the model being simulated.

It is not only the monitoring overhead that the real entities being situated on real physical hosts causes. When observed from the point of view of one real entity, the rest of the SIFSUT, which consists of the other real entities and the simulated components, may also be a source of overhead. We will refer this kind of overhead as *emulation overhead*. The effects of emulation overhead depend on how many physical hosts are participating in the experiment, and the distribution of execution over these physical hosts.

In the case of monolithic systems, or in other words an emulation-based experimental setting in-a-box, the emulation overhead may become one of the main factors preventing the system to scale. Even for non-monolithic, distributed systems, it is very desirable to keep the number of physical hosts in the experiment much lower than the number of real entities, or the nodes in the networks described in the scenarios. This is established by multiplexing the real entities on physical

hosts, for example by running more than one node of the network on one physical host. Such multiplexing should be done with the effects well accounted for, since usually the real entities create resource contentions and sometimes even conflicts. The most immediately noticeable resource contention that limits the scalability happens over the CPU: Vahdat et al. [180] nicely shows how the number of instructions an emulated node can compute per each byte it communicates, decreases with increasing number of emulated nodes per physical host. Furthermore, this also has a limiting effect on the aggregate throughput of all the nodes multiplexed on a host (see Figure 6 in [180]).

An initial estimation about the extent of the emulation overhead problem can be deduced from results reported in literature on the performance of the emulators themselves. Herrscher and Rotermel provide some data [82] from which it can be observed that their system induces around 8% difference between the specified bandwidth available to nodes in the emulation and their actual throughput, without pushing the system for scalability. Jain discusses in his book on performance analysis [90] that having around 10% effect on an observed variable affects the performance analysis results considerably. Therefore Herrscher and Rotermel's data can be interpreted as showing that further studies are needed for systematic analysis of these effects, assessment of their significance, and development of a methodology and a tool set in order for researchers conducting emulation-based experiments to actively manage these effects.

Chapter 4

MICA — A Software Architecture for Network Simulators and Emulators

In this chapter, we are going to present our approach for building network simulators and emulators, which we call the minimalistic component-based software architecture (MICA). We will begin by shortly explaining why we have felt that a new architecture is needed, and why we have chosen a component-based approach. Next, we will describe the component model of MICA, followed by examples on how to use the component model at the software design level, and how it supports simulator interoperability and model replacement for simulators that satisfies certain conditions. We have implemented two different component platforms for this model: one that provides only a single thread of control, and another one that implements the multitasking and multiprocessing support in the model by using a middleware called PVM. These will be described in the last section.

4.1 Motivation for a New Architecture

There are many different simulators and emulators reported in the literature, as presented in Sections 2.5 and 2.6, and in Appendix B. Taking into account that every software has an architecture, the question why we have decided to formulate yet another architecture needs to be addressed.

A very brief account of the initial efforts in this thesis would explain our motivation. What lead to the development of MICA started with our work on identifying the problems of network emulators, which was presented in Section 3.5. While working on this problem, we have observed that it was a desired property for the network emulators to have an architecture sufficiently similar to the nature of the network being modeled in the experiments conducted using them.

Our initial goal was to develop solutions for the problems we have identified, therefore we looked for a simulation system or library to use. We started working on the NS simulator [23], only to find out that its architecture is quite different than a network. Furthermore, its software architecture is very diffuse in its code, which is split between the object-oriented extension of the TCL scripting language (oTCL) and C++. In addition, we have observed that learning this system is not very easy and quite time consuming, which appears to be a qualitative observation shared by many in the networking community.

We then started looking into the architectures of other systems, and decided that they either do not satisfy our view of what it means to be component-based, or fail in satisfying the properties we

seek, which will be described in Section 4.3.1. We will explain in the next section our motivation in insisting on a component-based approach for network simulators and emulators.

4.2 Motivation for Choosing a Component-Based Approach

It was previously discussed in Section 2.7 that networks are perceived as composed of network components. Furthermore, we have identified in Section 2.4.2 the three conceptual levels where the concept of components show up in simulation. We believe that an approach that ensures similarity between the concepts of components in these three layers would appear to the experimenters as intuitive, thereby reducing the learning load. Combining these observations, we hypothesize that the modeling approaches and the software architecture of the simulators and emulators being built should be aligned to the component-based nature of the networks. Therefore, both models for network simulation, and network simulators and emulators should be organized as compositions of components.

At the software implementation level, it has previously been discussed in Section 2.3 that a component is a coherent and self-contained unit. In an unstructured system, the units are tightly entangled, which makes separate analysis of individual units infeasible and necessitates a global analysis [173]. Taking into account the discussion presented in Section 3.2 on the experimenter's goals and the limited available resources, it is clear that a time consuming global analysis is not feasible, either. Therefore, the architectures for network simulators or emulators should be organized as loosely coupled components that allows one to conduct analyses of units separately. It should be noted that such analyses of units we are referring to, are about correctness of the implementation, and not about model components level concerns such as accuracy or precision.

Another supporting argument for a component-based approach can be derived from the independent extensibility property, which we have presented in Section 2.3.1 as a characteristic that emerges in systems built using a component-based approach. Networks are composed of complex units. However, experimenters do not have the resources to implement from scratch models for all these complex units. Therefore, it is necessary and common practice that models of various units are used by different people than those who have developed and implemented them. As a result, independent extensibility appears as a characteristic provided by component-based approaches, and desired in network simulators and emulators.

In Section 2.3.3, we have identified four elements in component-based approaches: component models, component platforms, component-based architectures, and component-based frameworks. In this thesis, the focus is on the component models and platforms, not on component-based architectures and frameworks. The models and platforms have a strong influence on the alignment of the components at the three levels discussed in Section 2.4.2.

Focusing on component models and platforms, MICA provides the base part of software architectures for network simulators and emulators. It is not complete as a software architecture: simulation system or library developers need to extend MICA with suitable component-based architectures and frameworks. We will present in Section 4.4 that this particular focus enables MICA to provide the implementation level interoperability and model replacement in simulators and emulators under certain conditions.

4.3 Component Model

4.3.1 Desired Properties for The Component Model We Seek

We will start by describing the properties we decided to pursue in the component model we developed. These properties, each of which will be described further in this section, are:

- striving for completeness: structuring as much of a software system in terms of components as possible,
- supporting components with variable granularity,
- being simple and minimal,
- separating worker and run-time management components,
- supporting transparent distribution of components,
- and using messaging for component communication, and preferring asynchronous messaging over synchronous.

Striving for Completeness

One of the main trends behind the component-based approaches in general is the continuation of the modularity-oriented thinking in software engineering. However, as mentioned in Section 2.3.3 in relation to frameworks, and as can be observed in some of the component models reviewed in Appendix C, not every component model strives for complete partitioning of the system being implemented into components and a component platform.

Benefits of a complete partitioning of a system can be identified as the following:

Ensuring a common basis:
Using the component model for a complete partitioning of the whole system ensures that all resulting software modules share a common conceptual basis. A common basis applying uniformly throughout the system ensures that sources of dependencies between parts of the system are limited to those that are identifiable from the component model. Thereby a common basis can help both in integration testing and in understanding the units by eliminating or limiting hidden dependencies. This would be important for network researchers who are trying to understand and use components written by other developers.

Providing dual appearance for applications:
When a system is completely partitioned into components and a component platform, it can be viewed as a simple set of components, too. Such dual appearance makes it possible to use the component platform as a shared basis for interoperating two applications. In Section 4.4, we will exemplify how this dual appearance can be used for simulator interoperability and model replacement.

While we take the position that we should strive to have high completeness, ensuring absolute completeness as defined above is not yet possible today. The main method of software reuse on

today's operating system architectures is still using shared libraries of functions. Through the use of the functional libraries, various mechanisms defined in a component model can be rendered ineffective or irrelevant, thereby leading to situations where the goals of the component model being used are no longer satisfied. Therefore, how the operating system itself will be represented in a component model should be determined, and the components should strictly adhere to accessing the operating system through the mechanisms defined in the component model. One solution to this problem is creating wrapper components for every library the system depends on. Such a solution is easy to realize for a small number of dependencies, but can be tedious otherwise. To summarize, absolute completeness is a hard to satisfy goal.

Supporting Variable Granularity

In a software system whose design adheres to the completeness property, components of varying sizes might be needed. Being able to use components of varying sizes helps reusing components from other systems, as well as designing as much of the coherent parts of the system as possible into reusable components, small or large.

Designs composed of different component size distributions would have different requirements from the component models. If we make a rough categorization of components as small and large according to their sizes[1], two approaches are possible for managing these requirements. In one approach, the component model can be designed to satisfy the union of requirements for both small and large components. However, such an all encompassing set of requirements eludes a precise formulation. As an alternative, which we follow in our approach also, one can focus on the minimum necessary subset of the intersection of the requirements for small and large components. Then different and changing requirements can be satisfied through additional architectural layers or tiers that are built on the component model, and extend it. This approach is also in fine accordance with the simplicity and minimality properties defined below.

The requirement for supporting variable granularity coincides also with the need for modeling different networks or different parts of a network at varying levels of detail.

Being Simple and Minimal

The component model we are seeking should be simple enough to be usable by those whose major focus is not mastering the design and engineering of component-based software. Our target audience includes mainly network researchers. With regard to the discussion in Section 3.2, from time to time these researchers assume the role of the simulator developer as well as being the experimenter. Therefore, the model should empower the network researchers with the ability to reuse software built by others, while not requiring them to become professional software engineers. For this purpose, the component model should adhere to the KASAP principle: Keep it As Simple As Possible[2].

Minimality is our preferred method for ensuring simplicity. We would like a component model that describes components and their interaction in a minimal way. For this purpose, there should

[1] We acknowledge that this categorization is very rough, and size can be defined in various ways. However, our main argument can be easily modified according to finer definitions.

[2] The word "kasap" means butcher in Turkish, which coincides nicely with our preference of addressing simplicity through minimalism, where redundancies are chopped away as much as possible.

be as little redundancy as possible among the constructs and mechanisms in the architecture. If necessary, the redundant constructs and mechanisms can be realized as additional layers or tiers of architecture to be built using the component model.

The guiding principles we use for reaching minimality, can be formulated more precisely with two rules. First, exclusion of any construct or mechanism defined in the model must considerably affect the ability to attain one or more of the desired properties we are defining in this section. Our second rule for minimality is that any subset of constructs and mechanisms that can be implemented using the remaining constructs and mechanisms, should be left outside architecture.

With regard to network simulators and emulators, properties of simplicity and minimality help to ensure a steep learning curve for the component model and how to use the component platform that supports the components. This is accomplished by keeping the number of constructs in the model as low and the interactions between the constructs as simple as possible.

One particular decision that relates to the minimality property as defined here is whether to provide support for composite components in the component model. As discussed previously in Section 2.3.6, composite components have equivalents in flat component models. While the concept of composite components might provide better productivity to the developers at the expense of additional complexity that leads to longer learning time, they do not add to the representative power of a component model. Therefore, the case about composite components fits nicely to the second rule described above, and their exclusion from the component model does not hinder attainment of any of the properties we require. Thus support for composite components is regarded as a design tool issue, where a design tool may provide its user with an extended component model that provides composite components, and automate the mapping of composite components that have been built, into their flat model equivalents.

Separating Workers and Run-Time Management

The separation of workers and run-time management refers to our position that in the modularization of a software system, the modules that construct and manage the run-time structure of the system should be separated from the modules that implement the application logic. Our position is a slight modification of, and follows as a specific case from a position taken by researchers working on reconfigurable architectures. These researchers argue that separation of the dynamic re-configuration behavior from the steady-state behavior is possible and desirable [5, 8]. In a component-based approach with completeness as one of its design premises, this distinction can be realized in the final run-time structure of a system by using different components whose instances assume these different roles.

In order to accomplish the task of construction of the run-time structure of a system, a component requires access to certain functionality that needs to be provided by the component platform. These functions include

- creating and deleting component instances,
- managing the set of computing resources used by the system, such as functions for allocating or freeing processes or hosts,
- distribution of component instances over computational resources,

Table 4.1: Relationship between identification schemes and necessary functionality for construction and management of the run-time structure of a component-based system.

	Component IDs	**Component Instance IDs**	**Computational Resource IDs**
Component instance creation	X		
Component instance deletion		X	
Setting up component wiring		X	
Tearing down component wiring		Depends on existance of link ids with system-wide semantics	
Associating components with resources		X	X
Managing resources			X

- and setting up or tearing down the wiring between the components.

Such functionality requires the component to be able to recognize and use various identification and addressing schemes. Identifiers for components, component instances, and computational resources are necessary. The relationship between these identifiers and the functionality they are needed for is given in Table 4.1.

All these identifiers have the common property that in order to be useful, they need to have system-wide semantics at run-time: regardless of by which component instances, and in what part of their code an identifier of these types is used, all instances of the identifier should be pointing to the same entity in the system. Storage of these identifiers is frequently a necessity, whether directly as identifier instances, or as descriptions of what they are pointing to, which can be used with a known registry to query for the identifier.

Although these identifiers are necessary, storing identifiers that have system-wide semantics at run-time is a serious source of coupling. Such storage can be dynamic, such as by a component instance saving the identifiers it learned at run-time, or static where some of the identifiers might be integrated into the implementation of the component from which the instances are created.

Luckily, especially for our target domain of network simulators and emulators, construction of the system is done by a limited number of components. The other components, which we can refer to as workers, either do not need these functionalities, or they need them so infrequently that requiring these services from other "more capable" components should not introduce unjustifiable overheads. For the case of network simulators and emulators, dynamic re-configuration does not seem to make these systems able to simulate a wider range of models. As a result, the problem of couplings created by necessary use of identifiers with system-wide semantics can be confined to a small set of components. This can be established by separating the types of components that can construct and modify the run-time structure of a system, from the ones that actually carry out the

model logic.

A relevant question is whether and how much does the separation of workers and constructors conflict or support the simplicity and minimalism premises. In the component-based approaches, the components are expected to be independently developed. This implies that a developers that focus on development of an individual worker component might care for the context the component will be instantiated into, but at the same time they are indifferent to by which entity the instantiation is done. What resources a component instance requires from its instantiation context presents a slightly different problem. However, along with how the services provided by the component will be used, the description of required computing resources best fit into the contract that a component declares to adhere to. Therefore, from the point of view of the developers of individual worker components, separation of workers and constructors indeed serve to achieve simplicity. Minimality also follows from the unnecessity of including the functions described in this section to the API between the component platform and worker component instances. For constructor component instances, these functions are indispensable as we have explained, therefore such separation of components does not introduce additional complexity in constructor components, either.

Separation of constructors and workers also relates to the separation of the roles of simulation system and library developers, and experimenters, which were discussed in Section 3.2. Constructor components would mainly be used by simulation system and library developers, while worker components would simplify the model from the network researcher's point of view.

Lastly, in reference to terminology used by Allen et al. [5], separation of constructors and workers allows formulation of "pre-steady-state" re-configurations, as shall be exemplified in Section 4.4. How dynamic re-configurations can be supported, and what and whether additional conditions are necessary, is left as a future work.

Supporting Transparent Distribution

Ability to distribute the execution of the simulators and emulators onto more than one hosts is desirable and necessary in simulation and emulation-based network experimentation. Therefore the component model we are seeking should be able to allow component instances to be distributed over multiple hosts.

In addition, we would like such distribution to be transparent to the developers of the individual worker components. In our case, these developers are the network researchers developing their models of interest that they cannot acquire from other sources. The distribution of component instances is also related to a component's computational resource requirements, and contextual instantiation constrains such as being a singleton. However, as discussed previously, we regard these as contractual issues. Therefore, it should be possible to implement a component without caring for on which host and in what context the component will be instantiated, except for expressing such concerns in the component's required contracts, if necessary. In other terms, the component model should formulate resource allocation to a worker component as a set of operations independent from worker component's implementation. This means distribution of the components onto a set of hosts should be transparent to the worker components, and their developers.

Preferring Asynchronous Messaging

There appear to be two main approaches to providing abstractions for component instance communication at the component model level: using sets of operations at the endpoints, which are called interfaces, or using protocols defined in terms of messages along with simple message sending and reception.

Our position on this choice for the component model we seek is the use of asynchronous messaging based protocols for component communication, where protocols should be defined in component contracts. The reasoning behind this choice is readily very nicely explained by Szyperski in [173], pp. 155–156, and he even uses communication communities as the example for explaining the related factors:

> Sets of interfaces correspond to protocols. Instead of focusing on the required and provided operations at endpoints, protocols focus on the valid sequences of messages exchanged between these endpoints. It is always possible to rewrite protocol definition into a composed contract over sets of appropriately defined interfaces. The reverse is also true. When deciding which of these two dual approaches is best to take, a number of factors need to be considered:
>
> - *Tradition:* Many communications communities prefer messages and protocols while many computing communities prefer operations and interfaces. (This is changing as both communities grow together under the influence of the Internet.)
> - *Emphasis:* Protocols emphasize messages between endpoints, while interfaces emphasize operations on endpoints that accept and return messages.
> - *Synchronous v. asynchronous communication models:* Messages and protocols more naturally describe asynchronous communication, while interfaces an operations more naturally describe synchronous communication. (Asynchronous operations on interfaces can be defined and synchronous constraints in protocols are also possible.)

Although synchronous and asynchronous communication models would occur together in a network simulator or emulator, it is possible to define synchronous constraints in protocols, as Szymanski points out. Our position in this thesis is that tradition and emphasis as defined above have significant consequences on how intuitive the simulator's software architecture appears to the network researchers, who have to develop their models adhering to that architecture. Therefore, a component model that allows and prioritizes asynchronous messaging based component communications is preferred for our purposes.

4.3.2 Defining the MICA Component Model

We were not convinced that the component models surveyed in Appendix C satisfactorily supports the properties we seek in a component model for network simulators and emulators. Therefore, we designed a new component model as part of MICA. The differences between MICA and the existing component models are discussed in Chapter 6.

In this section, we describe the constructs in this component model. These constructs are summarized in Table 4.2, and explained in further detail below.

Table 4.2: Summary of constructs in MICA.

Construct	Short Explanation
Execution Managers (EM)	EMs represent independent computing resources.
Execution Units (EU)	EUs represent pseudo-independent computing resources.
Unit Models (UM)	Supertype of worker components (see Section 4.3.1).
Constructor Units (CU)	Supertype of constructor components (see Section 4.3.1).
Instances (UMIs and CUIs)	Component instances of UM or CU components.
Links	Unidirectional data pathways between component instances over which component instances communicate.
Message Types	A type system for messages need to be defined when developing component-based architectures on MICA.
Messages	Messages are used for communication between component instances.
Identifiers	EMs, EUs, instances, and links have associated identifiers.

Unit Model (UM)

Unit model is the supertype of all worker components in our component model. The name "unit model" was used since components of this supertype mainly corresponds to basic model implementations in network simulators and emulators.

As discussed in Section 4.3.1, the worker components need not have access to functionality regarding construction and management of the run-time structure of the system. Therefore, instances of UM type components are provided with services only for message reception, sending, and message type registration.

Constructor Unit (CU)

With regard to the worker-constructor separation, and in contrast to the UM, constructor unit is the supertype of all constructor components. Their instances have access to all functionality defined to be provided by the component platforms for MICA.

Instances

Instances are component instances created from UM or CU components. The instances created from UM components and CU components are called unit model instances (UMIs) and constructor unit instances (CUIs) respectively. CUIs are the instances that can create or delete other instances. The bootstrapping problem is addressed by requiring that a first CUI be created by the component platform, and a first message to be sent to that CUI as if this message was sent by the first CUI itself. Different component platforms may provide different methods for determining the CU component to be instantiated as the first CUI, and the type of the first message to be sent to it. Possible methods would be accepting human-readable form of their identifiers as command-line parameters or reading them from some configuration file variables.

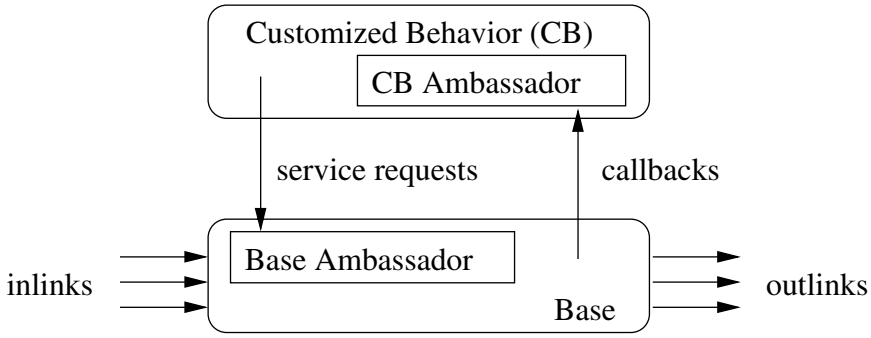

Figure 4.1: Logical structure of an instance.

As shown in in Figure 4.1, an instance is internally composed of two parts: a part we call the *customized behavior* (CB), and another called the *base*. The implementation of the base for the instances are provided by the component platforms. The CB is to be supplied by the component developer implementing the component's behavior. Both parts of the instance communicate with each other through the internal interfaces called the *ambassadors*. For those readers who are familiar with the IEEE 1516 High Level Architecture (HLA), the ambassador-based design in our component model is similar to the federate design in HLA, which is based on the RTI and Federate ambassadors.

The instances use the base assigned to them for invoking services from the platform implementation they are running on. Services are provided to the CB of an instance through the *base-ambassador* interface in defined in its base. Similarly, the base of an instance uses the instance's CB's *CB-ambassador* for making callbacks to the CB. Because of the worker-constructor distinction, the sets of services and callbacks differ for UMIs and CUIs, and so do the definitions of their ambassadors. Therefore, unless the ambassadors of both types of instances are meant, the UMI related ambassadors will be referred to with a *UMI-* prefix, as in *UMI-base-ambassador* and *UMI-CB-ambassador*, whereas the CUI related ones will be referred to with a *CUI-* prefix. The services and callbacks provided by these ambassadors are summarized in Tables 4.3, 4.4, 4.5, and 4.6. They are described in more detail in Appendix D.

It should be pointed out that being instances of worker components, the UMIs are not provided with any services that requires identifiers with system-wide semantics, as discussed in Section 4.3.1. The only identifiers UMIs can use are link IDs, whose semantics are defined only with regard to the UMI they are defined in, as will be discussed below. Therefore UMIs do not suffer from coupling caused by storing identifiers that have system-wide semantics.

The control flow for the instances is defined by the CBs running to completion in response to callbacks they receive through their CB ambassadors. The service requests from the base, which are invoked by calling methods in the base-ambassador, do not cause any re-entrant callbacks to the CB.

All instances can communicate by their CBs requesting a service from their bases to send a message to an outgoing link (see below) with a given link ID. In addition to using links, CUIs can also make use of the CUI IDs they already know, for sending messages directly to other CUIs. This additional mechanism is provided since CUIs already have access to the instance IDs. However,

Table 4.3: Services provided to CUIs through CUI base ambassadors.

Service	**Parameters**	**Short Definition**
Create CUI	CUI-CB type, where	Create a CUI with given `CUI-CB type` in the EU with id `where`.
Create UMI	UMI-CB type, where	Create a UMI with given `UMI-CB type` in the EU with id `where`.
Replace UMI-CB	UMI-CB type, umi-id	Replace the UMI-CB of UMI with id `umi-id`, with a UMI-CB of type `UMI-CB type`.
Create EM	EM descriptor	Create an EM described by `EM descriptor`
Create EU	where	Create an EU in EM with id `where`
Delete CUI	what	Delete the CUI with id `what`
Delete UMI	what	Delete the UMI with id `what`
Delete EM	what	Delete the EM with id `what`
Delete EU	what	Delete the EU with id `what`
Link	from, outlink, to, inlink, type	Create a link with id `outlink` at originating instance with id `from`, ending at instance with id `to` as link with id `inlink`, carrying messages of type `type`
Unlink	from, outlink, to, inlink	Delete the link described by 4-tuple (`from`, `outlink`, `to`, `inlink`)
Send Message to Link	outlink, msg	Send the message `msg` to outgoing link described with id `outlink`
Send Message to CUI	target-cui, msg	Send the message `msg` to CUI with id `target-cui`
Get EM List	none	Get the list of EMs set up in run-time
Get My Id	none	Returns the id of the CUI issuing this service
Register Message Type	msg-type	Make message type `msg-type` available to the calling instance

Table 4.4: Callbacks to be responded by CUI CBs through CUI CB ambassadors.

Callback	**Parameters**	**Short Definition**
CUI-Base Created	cui-base-ambassador	The base part of the CUI given as parameter `cui-base-ambassador` is ready to receive service requests
Receive Message From Inlink	inlink, msg	The message `msg` was received from the incoming link described with id `inlink`
Receive Message From CUI	sender, msg	The message `msg` was received from the CUI with id `sender`

Table 4.5: Services provided to UMIs through UMI base ambassadors.

Service	**Parameters**	**Short Definition**
Send Message	outlink, msg	Send the message `msg` to outgoing link described with id `outlink`
Register Message Type	msg-type	Make message type `msg-type` available to the calling instance

Table 4.6: Callbacks to be responded by UMI CBs through UMI CB ambassadors.

Callback	Parameters	Short Definition
UMI-Base Created	umi-base-ambassador	The base part of the UMI given as parameter `umi-base-ambassador` is ready to receive service requests
Receive Message	inlink, msg	The message `msg` was received from the incoming link described with id `inlink`

a CUI is not allowed to send a message to a UMI using the UMI's ID, since the mechanisms for reception would be different, and the UMI would not be able to know where the message is coming from.

Execution Unit (EU)

An execution unit (EU) is the abstract representation of a pseudo-independent computing resource. An example to such resources are the processes running in a time-shared manner on a single processor, managed by a multitasking capable operating system. Unlike components, EUs do not have types. They can be created or destroyed by the component platform on request from a CUI. A component platform would support creation of at least one EU. An EU is associated with a single execution manager (EM) (see below) at the time of its creation, and its associated EM can not be changed during the EU's lifetime.

Each instance in the run-time structure of a system is associated with a single EU, while an EU may be associated with multiple instances. For the sake of presentation, we will use the containment metaphor, and refer to the instances associated with an EU as being contained in it. An instance is associated with an EU at the time of the instance's creation, and the associated EU of an instance cannot be changed during its lifetime.

The main purpose of EUs, along with the EMs described below, is to allow construction of various control flow patterns in a controlled way. We refer to the part of the component platform that manages an EU as the *EU controller* for that EU. An EU controller receives messages arriving to the instances it contains in an asynchronous manner. The messages are then dispatched one by one to the instances in the EU, according to any precedence scheme that might be defined by the component platform. Since the callbacks defined in the CB-ambassador interface are defined to run until completion, the instances in an EU receive and process their messages in an exclusive manner. Therefore, EUs provide a multiprogramming environment for the instances they contain. The control flow relationships between the EMs, EUs, and instance are summarized in Figure 4.2.

Execution Manager (EM)

An execution manager (EM) represents an independent computing resource, such as individual computers or processors. Like EUs, EMs do not have types. They can be created or destroyed by CUIs, too. Naturally, what is being created or destroyed are not computers or processors, but controllers on computers or processors that allow them to be used as resources in the run-time structure of the system.

Continuing to use the containment metaphor in the sense it was used for EUs, an EM may contain zero or more EUs. Instances that are contained in the EUs contained in a single EM run in

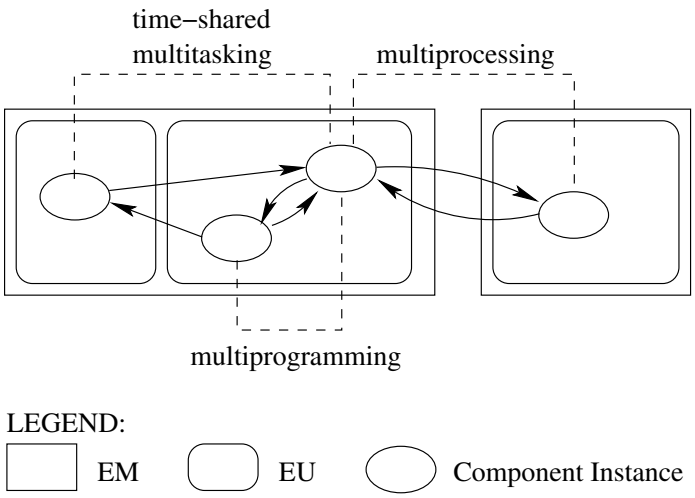

Figure 4.2: An example demonstrating how the EMs, EUs, and instances makes it possible to express multiprogramming, multitasking, and multiprogramming.

a time-shared manner, providing the means to express multitasking in our component model. In contrast, instances that are contained in EUs contained in different EMs run in parallel. Thereby our component model is also capable of expressing parallel processing relationships between the instances.

Links

Links are unidirectional data flow pathways between instances. As the links are defined to be unidirectional, every instance has two sets of links which will be referred to as the inlinks and outlinks. Each link is identified by a five-tuple that consists of the following:

- the ID of the instance the link originates from,
- a link ID that is unique among the link IDs of the outgoing links of the originating instance,
- the ID of the instance that the link ends in,
- another link ID that is unique among the link IDs of the incoming links of the instance the link ends in,
- and a message type.

Of the elements of this five-tuple, the pairs formed by the originating instance ID and the outgoing link ID, and by the ending instance ID and the incoming link ID, uniquely identify a link. Therefore from the perspective of a single instance, a link described by a defined link ID is a point-to-point communication medium with a fixed message type. The CB of an instance directs messages to an outlink using the outlink's link ID, and receives the inlink's link ID along with the messages received from an inlink.

The links can only be set up by CUIs, since their setup requires instance IDs. Instance IDs have system-wide semantics, therefore the UMIs are shielded from them. The CB of a UMI is only dependent on the link IDs, which have only semantics local to the UMI they are defined in. In effect, this reduces coupling and helps to prevent hyper-spaghetti of dependencies between instances.

The main purpose of including the message type in the link definition is that component platforms can make effective use of this information in transparently handling the data flowing on links that cross through the EU boundaries. This observation is based on our experience in the development process for the component platforms for MICA.

Message Types

Our work on the component platforms we have developed for the early forms of the component model indicated that an explicit type system for messages is needed. This is needed in order to provide transparent serialization for messages and dynamic loading of libraries that contain implementations of message classes. In particular, the C++ type mechanism is not adequate by itself for implementing object serialization. We consider C++ as an important language that should be supported.

Our component model requires every instance to register the message types it will be using, during the initialization of the instance.

Messages

The messages are serializable objects with associated message types. A component platform should not serialize and deserialize messages when the sending and the receiving instances are in the same EU. Message passing is non-blocking, that is sending a message does not block the sending instance. A message sent eventually arrives at its destination, unless the destination is deleted. The only guarantee on the message delivery order is that the messages sent by a sending instance arrives at the intended receiver instance in the order they were sent. Such a restricted guarantee on the order of delivery of messages is also used in the definition of the agent communication language KQML [58].

Identifiers

Different types of identifiers are used in the MICA component model to address different entities. Every EM, EU, instance, and link that is created at run-time have an associated identifier. The EM, EU, and instance identifiers only appear in the service calls defined in the CUI-base-ambassador and the callbacks defined in CUI-CB-ambassador. UMIs may handle them, such as by learning about them from a CUI through communication, but they are not provided with any services or callbacks that make use of these IDs. Link IDs have different semantics than the others, as explained before when links were being discussed. Furthermore, there are also identifiers for computational resources, components, and message types. These identifiers are more like descriptors, since they also need to be available for use by humans.

The EM, EU, and instance identifiers are opaque: their implementations are component platform dependent, and instances of these identifiers are created only through the component platform. While EM IDs are implemented as opaque, each EM ID is associated with a computational

resource ID, which is human readable. Such a human-readable form appears to be necessary since the developers of CU components and the users of the systems using these components would presumably need to be able to refer to the computational resources the components are to be distributed over.

In contrast to EM, EU, and instance IDs, the link IDs are not opaque: they can be cast to and from a set of values to be determined by component platform implementations, such as the set of natural numbers. Both CUIs and UMIs use link IDs to communicate through the links. The reason why the link-ids are not opaque, is that the link-ids of the links would be used in describing the behavior associated with an instance. Therefore they should be accessible to humans, such as by being casted to integers.

The EM, EU, and instance identifiers are defined to support equality checks, cloning, and serialization/deserialization. The serialize/deserialize operators are necessary for the instances to be able to convert the opaque ids to a form transferable in messages. Link IDs do not need to support serialization/deserialization, since they are not opaque. In addition to cloning and equality operators, the link IDs should support a smaller-than operator to allow for range checks.

4.3.3 On Leaving Out Simulation of Time

Simulated world is virtual and so it is subject to arbitrary rules set forth in a model. One typical set of rules relate to causal relationships between events, and the progression of time in the simulated world.

Simulation models for causal relationships and time are typically provided in the APIs provided by the simulation systems. There are various approaches being used in ensuring correct causal relationships and synchronisation, but typically a monolithic or distributed scheduler is used. Since these "schedulers" are implementing the concept of time as it appears in the virtual world, they are in fact sub-models of the model being simulated.

As with any sub-model in a simulation, model implementations that are simulating causal relationships and time should also be separated from software architectural concerns in a simulator. With respect to a component-based approach, this can be formulated by saying that such models should be provided as component-based frameworks that are to be used as an architectural tier in implementation of simulators. Simulation of time and causal relationships should not be a part of the component model, nor the component platforms. Therefore, MICA APIs do not contain any time management or synchronisation related APIs, but it can be used with different component-based frameworks implementing different time management and synchronisation methods, such as parallel simulation techniques including optimistic ones.

Leaving time management and synchronisation out from the MICA component model raises the question about how MICA can support component reuse in simulations. After all, time management is a typical dependency of a simulator component, and therefore MICA component model appear to lack a crucial component for ensuring reuse. However, it was presented in Section 2.3.8 that component reuse is closely connected with component-based frameworks and architectures. Therefore, providing only software architectural support for reuse at the component model level, and leaving time management and synchronisation issues to component-based frameworks to be used as additional architectural tiers is consistent with our reuse hypothesis presented in 2.3.8.

4.4 Usage Examples

In this section, we are going to describe how the MICA component model can be used in simulator development by looking at some hypothetical scenarios as examples. The concrete demonstration of the architecture is the network emulator DINEMO, which will be described in the next chapter.

We will start with describing the elements of a simple network simulator design. Then we are going to present a set of conditions for a simulator built using MICA, satisfaction of which enables the model replacement and simulator interoperability scenarios.

An earlier form of these examples, which used a bootstrapping method that we have abandoned in the evolution of the component model, was previously published in [72].

4.4.1 A Simple Simulator

The elements of a simple simulator are a CU component, a set of UM components, and a set of message type implementations that are used by the instances of these components.

The role of the CU component is to define how the run-time structure of the simulator is to be built. The instance of this CU component will have knowledge about which UM components to instantiate and in what patterns they should be linked. This knowledge may be incorporated into the code of the CU component, or its instance may be obtaining such information from outside sources, such as from a configuration file. As the first instance being created by the component platform, the CUI would then receive the start message. In response to this message, the CUI creates the other component instances, and builds links among the component instances. Then it starts the execution of the model implementations in some way, such as by sending a message to a certain UMI. These steps of execution for a simple simulator are summarized in Figure 4.3.

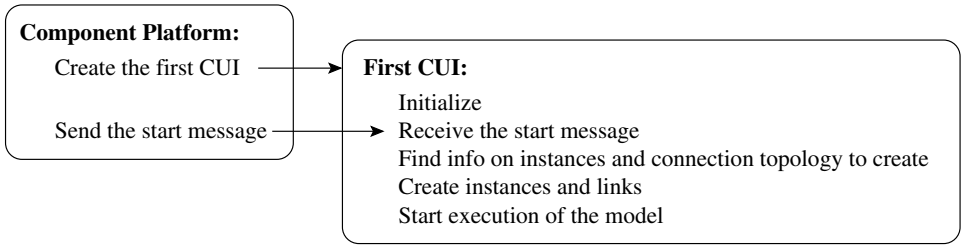

Figure 4.3: Steps of execution for a simple simulator.

The set of UM components correspond to model implementations that can be instantiated in building the stand-in for the system under test (SIFSUT).

As an example, consider a simple network simulator where the network model is partitioned into the following submodels: the communication medium C, hypothetical protocols P_L and P_H, where P_H is a higher level protocol working on P_L, a traffic generator T, and a sink S. The implementations of these models will constitute the set of UM components. In addition to these models, there will be a single CU component for the simulator, which we will refer to as CU. Furthermore, suppose that the scenario to be simulated involves two nodes connected to the communication medium, where each node is composed of an instance of T and an instance of S connected to a P_H, the P_H connected to a P_L, and the P_L connected to the C. The communication

medium C would be instantiated as a singleton. We will further assume that the simulation is started by sending start messages of type t_s to the traffic generators, and the CU expects s as its first message. Then, the steps up to the start of execution of the simulation episode are as follows (see also Figure 4.4):

1. The component platform is invoked with CU as the component to create the first instance from, and message type s as the first message to be sent.
2. The CUI is created from the CU.
3. The CUI receives a message of type s.
4. In response to s, the CUI creates the UM component instances and the links that define their communication paths in accordance with the scenario.
5. It creates and sends t_s messages to the traffic generators, in order to start the execution of the episode.

At the end of the episode, a stop message may be sent to the UMIs, followed by their deletion by the CUI. The component platform can then be stopped.

4.4.2 Some Conditions for Interoperability and Model Replacement

In Sections 4.4.3 and 4.4.4, two scenarios that demonstrate model replacement and simulator interoperation will be described. For these scenarios to work, certain conditions should be satisfied by the simulators that will be used. These conditions are presented in this section. A summary of these conditions is shown in Figure 4.5.

The first condition to be satisfied is that the creation of the stand-in for the system under test (SIFSUT), by instantiating UM components and setting up links, should fully precede the execution of the episode. New instances or links must not be created during execution. Although this condition appears to be quite restrictive, situations where the experimenter can not predict an upper bound on the number of model instances, are rarely encountered in network simulations.

While it is possible to employ multiple instances of multiple CU components in the run-time structure of a simulator, the first CUI to be created is an easily locatable single point of contact for the whole simulator. As the second condition, we will require that this first CUI for the simulator should support two services that can be invoked using messages sent to it. The first service should construct the SIFSUT, and the second service should configure the individual UMIs, and start the execution of the episode. We will refer to the messages for invoking these services as `CONSTRUCT`, and `INIT-AND-RUN` messages.

The third condition is closely related to the second one: The first CUI should send first a `CONSTRUCT` and then an `INIT-AND-RUN` message to itself when it receives the first message, which is injected by the platform for bootstrapping. Other than sending these two messages, the first CUI should not do anything in response to this first message. Instead, all relevant operations should be triggered in response to these two messages only. This presents the only difference in the sequence of events that happen prior to the start of execution of the episode, between the sequence of events in the simple simulator discussed in the previous section and in a simulator that adheres to the conditions in this section.

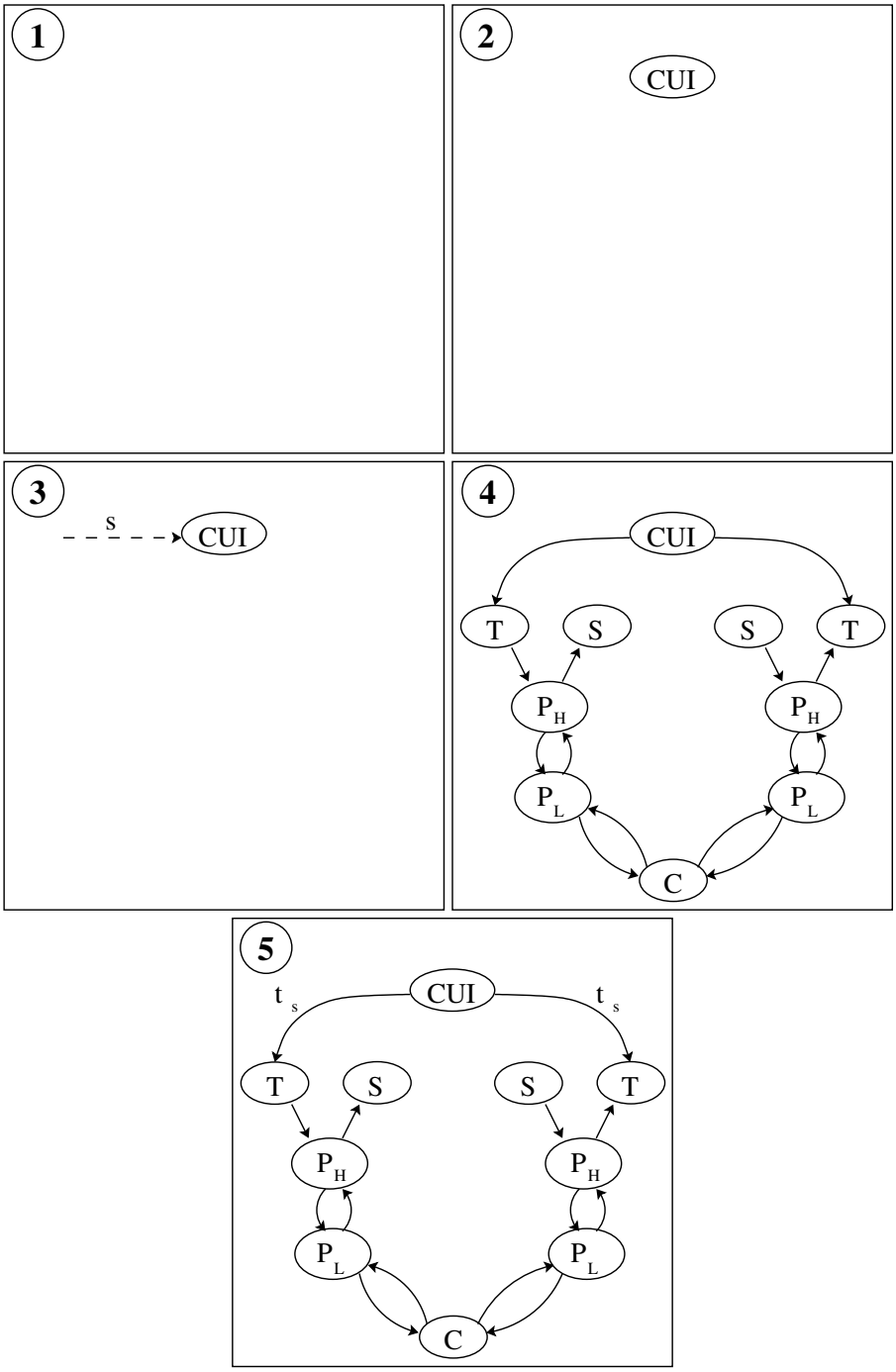

Figure 4.4: The steps for the example simple simulator, with two nodes.

1 Creation of SIFSUT should precede execution of the episode

2 The first CUI should provide these services:
- CONSTRUCT: Prepare the run−time structure
- INIT−AND−RUN: Initialize instances and start episode execution

3 In response to the first message, the first CUI should
- send CONSTRUCT and INIT−AND−RUN messages to itself
- not do anything else

4 The first CUI should be able to provide
- a mapping between names that are based on roles to UMI IDs
- information on the connection topology of the UMIs

Figure 4.5: Conditions that should be satisfied by a simulator for the simulator interoperability and model replacement scenarios to work.

The fourth and last condition is that there must be a way to obtain the IDs of the UMIs created in the run-time structure after the construction phase is carried out in response to a `CONSTRUCT` message. This method should be based on symbolic names that relate UMIs to the role they play in the simulation model. In addition, we will require that the first CUI should be able to reply to queries for mapping these symbolic names to UMI IDs. The reason behind this requirement is that for model replacement, one should first be able to locate the UMIs that implement the part of the model that will be replaced. Simulator interoperability is similar, where again certain UMIs are replaced and links are set up. This name to ID mapping can not be provided by any component platform alone. While the platform can provide information about from which components the component instances are created from, this is not enough for finding out about the role of a particular UMI in a simulation model. Information about such roles can rarely be deduced solely from how the UMIs are connected to each other.

In addition to the symbolic name to UMI ID mapping, the first CUI may provide information about the interconnection pattern of the UMIs. This might be necessary in some approaches to model replacement for correctly accounting for the connections of the to-be-substituted set of UMIs to the rest of the component instances in the run-time structure.

4.4.3 Example Scenario for Replacing Models

This scenario demonstrates two related situations in one scenario: replacement of simulator components (see Section 2.4.2), which implement a part of the model being simulated by a simulator, and forced use of a simulator in an emulation-based experimental setup. The scenario we present in this section is illustrated in Figure 4.6.

Suppose that there is a simulator, which we will refer to as *SIM*, whose architecture is based on MICA. Suppose further that there also exists a program and an associated protocol stack, which have been virtualized on a particular machine architecture and operating system. We would like

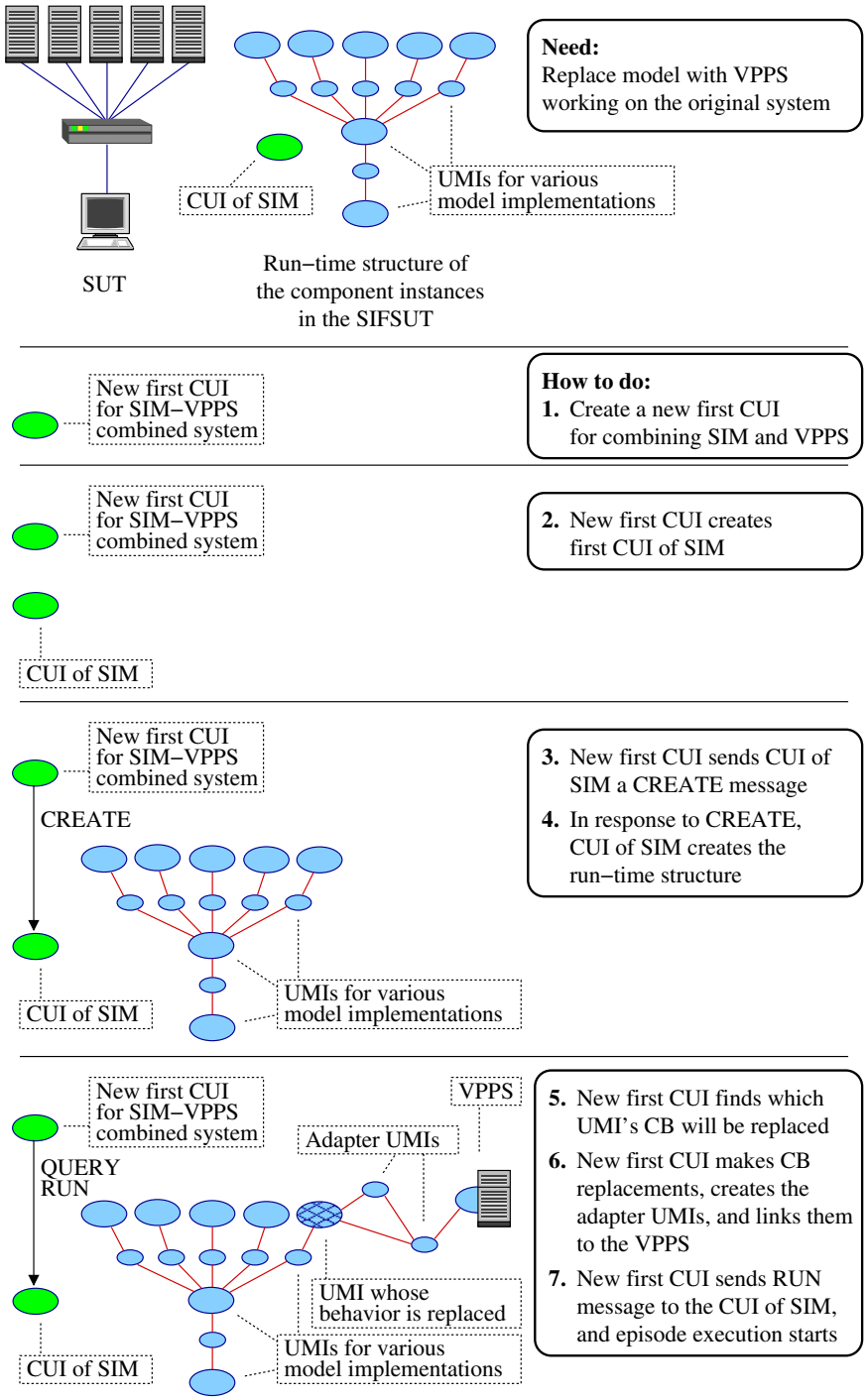

Figure 4.6: Summary of the model replacement scenario.

to use this virtualized program and protocol stack (VPPS) as part of the stand-in for system under test (SIFSUT) in an emulation-based experimental setup, using SIM as the emulator. The machine architecture or the operating system the virtualization of the VPPS was made in, can be different from the machine architecture or the operating system SIM was designed to run on. If they are different, a distributed component platform for MICA that works on both machine architectures and operating systems should be available.

A reminder of caution is necessary about forcing SIM to work as an emulator. Using simulators in emulated settings is a source of potential problems not necessarily addressed in simulator designs, as discussed in Section 3.5. While it is not our intention to discuss emulation related potential problems with respect to this particular scenario, we will provide a single example: about time, we note that either

- the simulator SIM must be capable of managing a virtual time that is synchronized to the wall-clock time,
- or the VPPS must be capable of virtualizing the time as perceived by the program and protocol stack,
- or the experimental goals should not be affected by the inaccuracies that might be caused because of time incoherence.

In order to integrate the VPPS with SIM, one must first define one or more UM components, instances of which will logically encapsulate the VPPS in the run-time structure. These UMIs will enable the VPPS to interact with the component instances in the run-time structure of SIM. For the sake of simplicity, we will assume in this scenario that the VPPS will be represented by a single UM component, which we will refer to as VPPS-UM component. When the VPPS is represented by more than one UM components, it is possible that there will be a CU component for overseeing the construction of these UMIs and the links between them. In that case, the situation will resemble a bit more to the simulator interoperation scenario that will be described in the next section.

In order to integrate a VPPS-UM component instance (VPPS-UMI) into the run-time structure of the model being simulated in SIM, we need to have a new CU component. This new CU component would be instantiated to create the first CUI for the combined SIM and VPPS-UM system, instead of the first CUI of SIM. If this new CU component is also implemented in a way that it adheres to the conditions set out for simulators in the previous section, it becomes possible to replace models in the combined SIM and VPPS-UM system using the same techniques described for simulator SIM in this scenario.

The customized behavior (CB) of the new CU component should:

1. construct the run-time structure of simulator SIM,
2. find and replace the CB of certain UMIs with either the VPPS-UM component's CB, which will construct or invoke the virtualized program and protocol stack, or a UM component CB which will forward messages back and forth between a VPPS-UMI and the rest of the SIFSUT,
3. and construct additional UMIs and links that will transform or transfer messages to be passed between the component instances in the SIM and VPPS-UMI, if necessary.

To accomplish the first task, the new first CUI will first create the first CUI of SIM. Then it will send the first CUI of SIM a `CONSTRUCT` message. The first CUI of SIM knows about the `CONSTRUCT` message since we assumed that it satisfies the conditions presented in the previous section. In response to the `CONSTRUCT` message, the first CUI of SIM will construct the run-time structure from its perspective. The only difference from the point of view of the first CUI of SIM, is that it does not receive the first message sent by the component platform.

The new first CUI will then create the VPPS-UMI, using a `CreateUmi` service request. The VPPS-UMI may construct the necessary environment related to virtualization of the program and protocol stack at this point in execution, or it may defer such initialization until the simulation model is initialized by the first CUI of SIM.

The third task that the new first CUI must do, is to find and modify the UMIs whose role will be assumed by VPPS-UMI. In order to find the IDs of the relevant UMIs, and how they are connected to the other UMIs in the run-time representation of the model to be simulated, the first CUI must query the first CUI of SIM. Supporting such queries has previously been described as another condition SIM should satisfy.

Once the first CUI identifies the relevant UMIs, it issues `Replace UMI-CB` service requests to replace the customized behavior on these UMIs, with behaviors that will take role in channeling messages back and forth between the component instances in the run-time structure built by SIM, and the VPPS-UMI.

An apparent alternative to using the `Replace UMI-CB` service request, is to delete the relevant UMIs in the run-time structure, then creating new UMIs with necessary behavior, and finally linking these new UMIs to the run-time representation of the simulation model in exactly the same way as the deleted ones. The reason why this would not work is that information about UMIs' identifiers can be stored in two places in a system built on MICA: in records kept by the component platform of the links that are set up between the UMIs in the system, and in the CB's of CUIs. Although we update the information about the deleted UMIs when we link the newly created ones in exactly the same way the deleted ones were linked, no assumptions were made that the first CUI of SIM be the only CUI in SIM. Furthermore, we have no control over the data structures of any CUIs except for the new first CUI instantiated from the CU component created specifically for combining SIM and VPPS-UMI.

Unless additional conditions are imposed on the first CUI of SIM for updating its and any other CUIs' data structures in case of such UMI replacements, the replacement strategy results in invalidation of the deleted UMI's identifier stored in CUIs. To prevent that, the new first CUI must use the `Replace UMI-CB` service, as this service does not change the UMI-base of the UMI it is invoked on. This means that the UMI identifier of the UMI that is updated to have a new behavior, does not change. Thus no data structure kept by CUIs is invalidated. Furthermore, the `Replace UMI-CB` service request does not even necessitate re-creation of the links connected to the UMI the behavior of which is being replaced.

The fourth and last task the new first CUI must do is to establish the connection between the VPPS-UMI and the UMIs whose behavior were replaced, if VPPS-UMI was not created as replacing one of the UMIs. Depending on the design, this may involve simply linking VPPS-UMI and the UMIs whose behaviors were replaced, or it may require creation of some number of other UMIs which would adapt the behavior of the VPPS-UMI to the behavior expected from the UMIs whose behaviors were replaced.

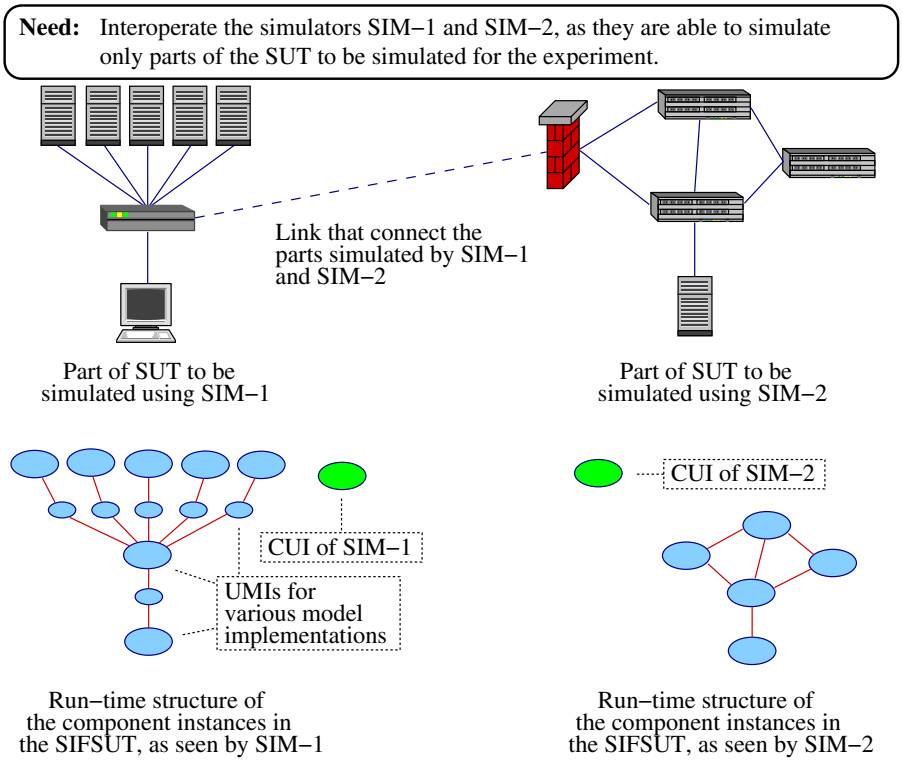

Figure 4.7: An example to the problem setting for the interoperation scenario.

At this point, the run-time structure created by simulator SIM is extended with the VPPS-UMI, and it is ready to be run. The new first CUI sends the `INIT-AND-RUN` message to the first CUI of SIM, and the episode runs. When the episode is finished, the new first CUI will have to do necessary clean-up tasks for the UMIs that the first CUI of SIM is not aware of, such as the VPPS-UMI and the UMIs that were used as adapters between the VPPS-UMI and the component instances created by SIM.

4.4.4 Example Scenario for Simulator Interoperability

In our second scenario, we will discuss how two simulators, each of which is designed on MICA, can be interoperated. We will not give a detailed step-by-step description as was done in the previous section, but we will focus on differences from the previous scenario. The scenario described in this section is illustrated in Figures 4.7 and 4.8.

Let us name the two simulators we have in this scenario as SIM-1 and SIM-2. In a normal stand-alone execution, these simulators adhere to the conditions described in Section 4.4.2. We need a new first CUI for the combined SIM-1 and SIM-2 system, as it was in the case in the model replacement scenario in the previous section. This new CUI C of the combination will create the first CUIs of SIM-1 and SIM-2, in the same way the first CUI in our previous scenario creates the first CUI of SIM. C will request both CUIs of SIM-1 and SIM-2 to create the set of component

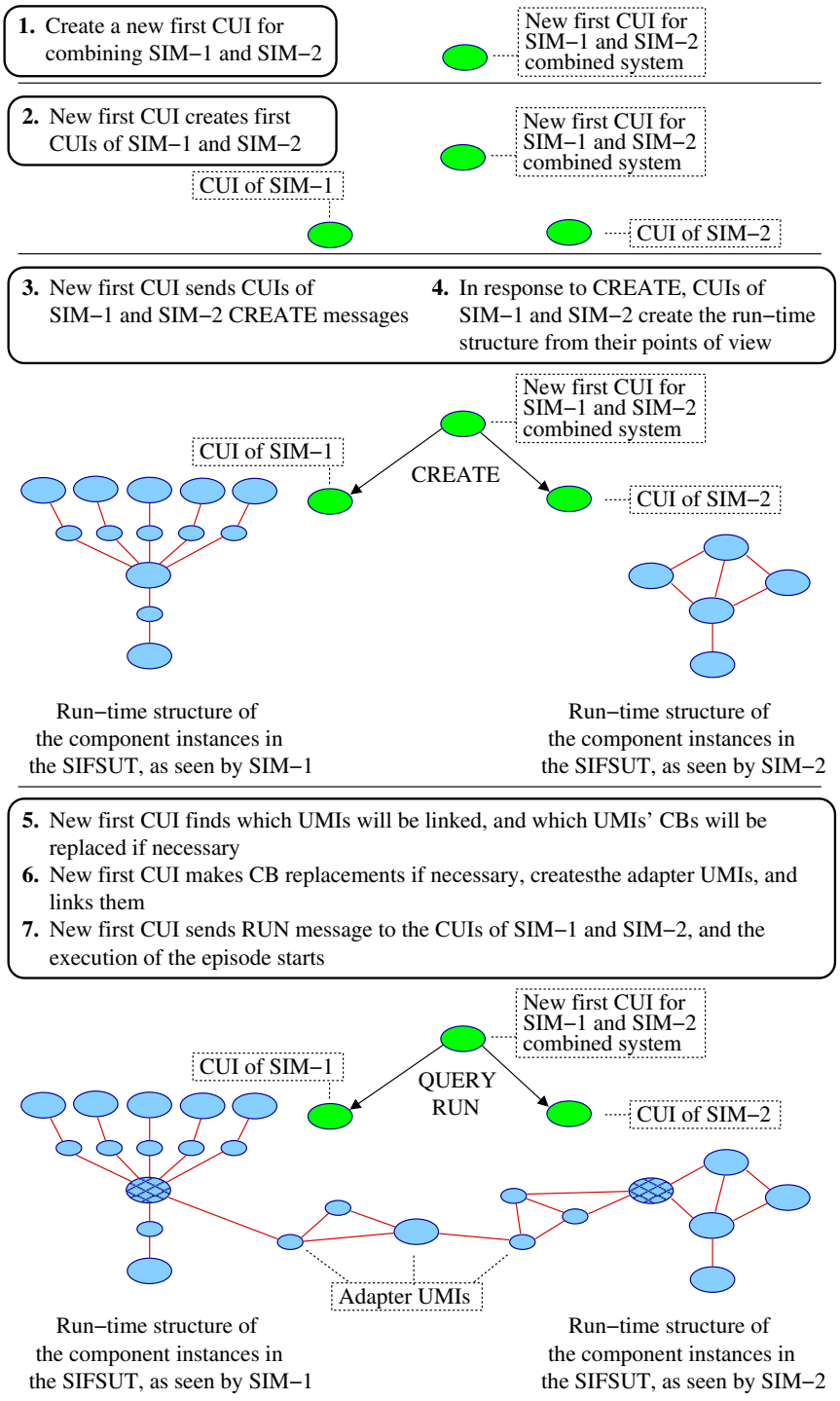

Figure 4.8: Steps for interoperating two simulators.

instances that constitute the stand-in for the system under test (SIFSUT) from their point of view. When creation is complete, C will find and modify certain UMIs created by simulators SIM-1 and SIM-2. Such modification will be done by replacing their customized behavior, again as discussed in the previous section. C will then create the necessary adapter UMIs, set up their links, and send the `INIT-AND-RUN` messages to the CUIs of SIM-1 and SIM-2, starting the execution of the episode.

Although the mechanism for interoperating two simulators at the level of component instances in the run-time structure, can be described with relative ease, such interoperability is not the only issue. The exact behaviors needed from the adapter UMIs, and how their connections should be set up, is a problem whose solution depends on various design details of the simulators involved. With regard to the levels the concept of components are used in simulation (see Section 2.4.2), the scenarios we describe here demonstrate a part of a solution at the simulator components level. It must be noted that this solution is not intended to simultaneously provide a solution at the model components level as well, and the solution to that problem is out of our focus in this thesis.

4.5 Component Platforms for MICA

4.5.1 Design Goals and Decisions

We start our presentation of our component platform implementations for MICA by presenting the design goals and various design decisions that have been influential. These design goals are closely related to the desired properties for the component model, which were presented in Section 4.3.1.

Isolation of Instances

Ideally, the component instances making up a system should only be able to communicate among themselves through the component platform. However, it is difficult to prevent component instances from using various libraries to create alternative communication mechanisms that bypasses the component platform.

Therefore a weaker form of isolation is set as a goal for the component platforms we have implemented. In this weaker form, we only require that it should be difficult and discouraging to implement components that refer to the symbols defined in other components. Therefore, the components should not be able to easily call functions or access variables defined in other components at run-time. Such direct symbolic references create dependencies buried deep into the code of the components, thereby considerably increasing coupling, and complicating component reuse.

Support for Small Size Components

It is not easy to provide a scale for component size. However, in reference to an intuitive use of the concept of component size, it can be said that system designs composed of different component size distributions would have different requirements from the component platforms. For example, a system containing a large number of small size components would need a component platform capable of providing low-overhead communications.

We believe that the components used in network simulators and emulators can mostly be regarded as small size components. Therefore, we have been careful to employ communication techniques that do not introduce unjustifiable overheads in our component platform implementations. However, we have not attempted to optimize the implementations of the platforms for minimization of these overheads, since we do not see optimization as a goal for our component platforms. These platforms are implemented mainly to serve as proof of concept demonstrations for our component model.

Costs of Management of the Run-Time Structure

Separation of workers and constructors has implications on how the efficiency concerns are to be addressed in component platform implementations. In reference to our argument that the functions for managing run-time structure of a system are needed only by a small subset of component instances in the system, it should be safe to assume that these functions are invoked relatively infrequently.

As a result, the implementations of these management functions should not be a major focus of efficiency concerns. Therefore, we deem sensible increases in the cost of these functions as a result of efficiency concerns related to worker components as tolerable.

Choice of Programming Language

The component model of MICA is programming language independent. For component platform implementations, we have chosen to use C++. It is chosen because it is an object-oriented language, while at the same time it natively supports C. C can be considered to be the foremost system programming language, therefore it was perceived as necessary to support C in a network simulator or emulator architecture. Using C code or code fragments might be beneficial in constructing models from protocol implementations.

The C++ API for the MICA component model is included in this thesis as Appendix E. Both of the component platforms we have developed, the single threaded one and the PVM based distributed one, share this C++ API. This allows component implementations written using this API to be used with any of the two platforms.

On Using XML

There appear to be two alternatives for implementing messages and message types: using an object-oriented approach, or using XML.

Using objects as messages has the potential risk of programmers making use of behavior defined in the message class, in addition to the message data. This would result in increasing components' dependencies on the message class implementations. Furthermore, developers can easily be tempted to incorporate object references into the message objects. Objects pointed to by such references may break down the system if they start to be shared between components. In addition, transparent serialization is not easy to achieve, if not impossible, for message objects containing references to arbitrary objects, in some environments such as that provided by the programming language C++.

Since what XML does is just describing data syntax, it does not introduce such a risks. Furthermore, the need for transparent serialization and deserialization can readily be addressed by using schema descriptions.

However, introducing XML into component platforms results in additional learning load not only on the simulator developers, but more importantly on the model developers as well. In network simulation, experimenters frequently act as model developers. Since one of the goals of MICA is to provide as little extra load as possible on the experimenters, requiring them to learn XML was considered an extra load, not justified enough even given the advantages.

In addition, it was considered that XML might introduce performance penalties, and may prevent any no-copy mechanisms to be implemented. Further research is needed that address the feasibility of using XML efficiently in MICA component platforms.

4.5.2 Single-Threaded Platform (RTI-st)

The initial design of the MICA was published in [70] under the name AMINES-HLA. The work on the single threaded component platform, which we also refer shortly as RTI-st (Run-Time Infrastructure – Single Threaded), started with that first version of the design. The RTI-st have been updated as the architecture matured further with the inclusion of constructs for distribution, such as EUs and EMs.

Control Flow: Meaning of Being Single Threaded

Single threaded means that there is only a single thread of control. In terms of the constructs in the MICA component model, being single threaded means that there is a single EM that contains a single EU, which contains all the component instances in the system. Therefore RTI-st supports only multiprogramming.

While RTI-st is usable with many component-based system designs, there exist some systems that cannot be run on RTI-st. One class of such systems form a subset of component-based designs that employ the singleton component instantiation pattern. Components can be instantiated as singletons in an EU, singletons in an EM, or as a system-wide singleton. When the design of a system requires more than one instance of a component to be instantiated as an EU or EM singleton, the system has to be distributed over multiple EUs or EMs. Therefore for running such systems, RTI-st can not be used.

Even if the target deployment environment of a system that is being developed is distributed, using RTI-st might be beneficial for testing and debugging. There are two reasons for this. First reason is that in RTI-st, the sequence of events happening in the system is independent of the relative running speeds of the hosts or processors, or the scheduling of processes by the hosting operating system. The second reason is that when events that happen in the system are logged along with system time in the RTI-st, the system time provides a complete ordering of message processing.

Implementation of Identifiers

Implementation of the identifiers can be divided into three groups: descriptor style IDs, link IDs, and opaque IDs.

The descriptor style IDs are computational resource IDs, component IDs, and message type IDs. Of these, computational resource IDs are irrelevant to RTI-st, since the only computational resource for the RTI-st is the host and the process it is running in. The component IDs are imple-

mented as simple strings, since they need to be readable by humans. Message type IDs also need to be readable by humans, but additionally an equality operator is required for the message type IDs, to be used in the CBs of instances for determining the type of incoming messages. Therefore message type IDs appear in two different forms for two different uses: while they appear as simple strings for the purpose of message type registration by the CBs of instances, they are used as opaque objects hidden behind the `MessageType` interface in the C++ API for the purpose of type checking in the CB implementations.

While the link ID is implemented as a class in the RTI-st, direct use of instances of this class in the customized behavior (CB) parts is allowed. New instances of the link ID class can be created using the `new` operator, or by casting from an unsigned integer. Casting instances of the link ID class back to an unsigned integer is also supported.

The opaque IDs, namely the EM, EU, and instance IDs, are exposed to the CBs of CU components only as abstract classes. The implementation of these IDs are hidden behind these interfaces, and the CBs of CUIs can only use pointers to the objects that implement them. While cloning is supported by a method defined in these interfaces, the CBs can not use the operator `new` to create new instances of these IDs. The factory pattern that is used for creation of EMs, EUs, and instances, also serve to create new instances of EM, EU, and instance IDs.

While the CBs of CUIs are denied access to the implementation and contents of the opaque IDs, they might need to communicate IDs with other CUIs. For such communication, serialization and deserialization of the opaque IDs need to be supported. Serialization is presented to the CBs as a normal method of the abstract classes implementing the opaque IDs. However for implementing deserialization, a static method is defined in these abstract classes.

Implementation of Message Passing

Message passing is implemented as pointer shuffling. Therefore messages are never copied in the RTI-st. Such a no-copy mechanism is in fine accordance with the goal of supporting small components in an efficient manner.

However, no-copy mechanism also has a drawback. When sending a message, the resources for a message are allocated by the sending instance's customized behavior (CB), and the control of these resources are then first transferred to the RTI-st via one of the message sending services, and then from RTI-st to the receiving instance's CB via one of the message reception callbacks. Naturally, the receiver of a message may forward it if it chooses to, in which case the control of the resources further changes hand. Reuse of resources allocated for a sent message is a potential source of defects about which the developers of instance CBs need to be careful.

Implementation of EU Controller

The main module in the implementation of the RTI-st is the EU controller. It keeps track of which instances can be activated, and controls the message dispatch.

Each instance has an associated FIFO queue for received messages. An instance can be in one of two states, blocked or ready for activation, depending on whether there is at least one message in its reception queue. The lists of blocked and ready instances are kept in the EU controller, and updated according to new messages being sent by the instances.

Instances that are ready for activation are activated in a round-robin fashion. When an instance is activated, all messages in its reception queue at the time of activation are dispatched to it one by one, by calling the appropriate callback function of the CB-ambassador of the instance.

Use of Dynamic Loading and Linking

Normally, the code of a process is loaded and the linker is run to bind symbols to memory addresses, before the process starts executing. In dynamic loading and linking, pieces of code are loaded and linked after the process starts, on request from the code executing in the process. The loaded pieces of code are referred with different names in different systems, such as dynamically loaded libraries (DLL), or dynamic shared objects (DSO). We will follow the naming in the GNU/Linux systems and use DSO, since RTI-st is implemented on GNU/Linux. Detailed information about loading and linking can be found in [51] and [110].

Component and message implementations are dynamically loaded in RTI-st. This way, component and message implementations all exist as separate DSOs, all separately compiled. Packing each component implementation into its own DSO, along with using opaque identifiers whose implementations are hidden, makes it discouraging for the component developers to build two components that depend on direct method call or direct variable accesses between their instances. In the case of messages, the symbols in the class that implements a message are made known to the components whose implementations make use of message objects of that class, through use of a header file that declares the message class. Using DSOs ensure that implementation of a message class comes from a single source. Otherwise, there exists a possibility that different implementations of the message class, with identical interfaces, might be compiled into each component implementation.

It is acknowledged in the developer community that use of dynamic loading and linking along with C++ becomes problematic if run-time type information (RTTI) or exceptions that traverse DSO boundaries are used. Furthermore, different C++ compilers uses different application binary interface definitions (ABIs), which in particular means that symbol name mangling and exception handling are not portable at the binary level between binaries produced by different compilers.

In fact, one of the reasons for inclusion of the discussion of a type system for messages into the MICA component model have been the difficulty in using RTTI with dynamic loading. The main and immediate need in the MICA component model is a set of types along with a suitably defined equality operator. Such a need is easily addressed by implementing a basic type information system instead of using RTTI, where all messages have a method that reports its type. In this way, the costs associated with RTTI are also avoided.

Throwing exceptions is a useful method for expressing control flow in the case of errors. In the implementation of RTI-st, they could have been used, and indeed tried, in defining the error conditions for the methods in the C++ API. However, calls to methods defined in the C++ API crosses DSO boundaries quite frequently. Therefore, in order to avoid the problem about throwing exceptions across these boundaries, error conditions had to be represented by return values instead of exceptions thrown. This aspect of the C++ API for MICA, can be contrasted to the federate APIs in IEEE's HLA standard, where exceptions are used.

Symbol name mangling has a simple solution: avoid the C++ compiler's symbol naming scheme. C++ compilers support definition of C style symbols, which appear in the compiled DSOs as

unmangled. This solution works well for finding the factory functions in the DSOs. However, in order for the component platform to be able to call methods in the classes defined in the DSOs that contain component and message implementations, the DSOs and the EU controller that is used for loading them should be compiled with the same compiler, or compatible compilers. However, it would be possible to use component instances compiled using different compilers, by placing them into different EUs that are compiled with the same compiler as the components.

The exact sequence of events that take place during dynamic loading of message and component implementations will be described in Section 4.5.4, along with how messages and components are implemented by developers.

Invoking the Platform

A system built on the RTI-st is started by invoking the RTI-st at the command line, and providing it with the component identifier for the first CU component to be instantiated, and the string form of the message type identifier of the first message to be sent to it. The first CUI receives this first message as if it is sent by itself, targeted to its own instance identifier. The available command line options for the RTI-st are given in Table 4.7.

Table 4.7: Command line options for RTI-st.

Short Form	Long Form	Explanation
`-l`	`--log=` *<FILE NAME>*	File name for the log file, where the messages from the RTI-st itself will be logged.
`-v`	`--loglevel=` *<LEVEL>*	Logging level for the messages from the RTI-st itself. Can be one of `ALL`, `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `OFF`.
`-o`	`--object-dir=` *<DIRECTORY>*	Directory which holds the component and message implementations.
`-c`	`--first-cui=` *<COMPONENT ID>*	The component identifier of a CU component to be instantiated as the first CUI in the system.
`-m`	`--first-msg=` *<MESSAGE TYPE ID>*	The string form of the message type identifier which determines the type of the first message to be sent to the first CUI.
`-h`	`--help`	Print help on command line options and exit.

Execution of RTI-st ends when there are no more messages in any of the receive queues of the instances in the run-time environment.

Unimplemented Parts of the Component Model

The `Replace UMI-CB` service is currently not implemented in the RTI-st. The implementation of this service is straightforward, and it was not regarded as immediately necessary for testing the component model.

Since the RTI-st is single threaded, it does not allow new EMs or EUs to be created, nor the single EM and EU be destroyed. Therefore the respective service requests returns with an error to the caller.

4.5.3 PVM-Based, Distributed Platform (RTI-PVM)

Parallel Virtual Machine (PVM)

Parallel Virtual Machine (PVM) [68] is a messaging based middleware for distributed scientific computing. The first version of PVM was developed in 1989, therefore there exists a significant community using it. PVM supports various platforms, including multiprocessor ones.

Our work on a distributed component platform started with a design that is built directly on the socket API. Then, because of the reasons discussed below, we have abandoned direct use of socket API and started using PVM.

Motivation for Choice of PVM

There are two primary reasons for choosing the PVM for the first distributed component platform implementation for MICA: PVM has some entities that can be mapped more or less directly to the entities in the MICA component model, and some of the functionality for implementing the component platform can be obtained as services from PVM libraries. We will shortly mention the relevant features of PVM in this section, leaving a slightly more detailed explanation of how they are used in implementation of constructs in MICA to the sections that follow.

One of the entities in the PVM that can be mapped onto MICA component model constructs is the `pvmd` daemon that runs on all of the hosts participating in the parallel virtual machine. The `pvmd` daemons, along with the processes they spawn on the hosts, provides us with a natural way of expressing the EM and EU constructs in the MICA component model. PVM also provides identifiers for the hosts and processes spawned, which provides support in implementing EM and EU identifiers.

Another useful feature of PVM for our purposes is that PVM provides ready-made well tested message passing primitives that can work across different computer architectures and networks, which we can use to implement component communication.

Furthermore, while we have not used them in this implementation, PVM also provides features such as debugging and resource managing tasks, which can be used to further develop component platforms that provide better support for debugging and that addresses load balancing issues for network simulators. Therefore using PVM provides room for improvement of the component platform using its advanced features.

Before deciding on PVM, we also had a look at the Message Passing Interface (MPI), which is also a popular middleware for scientific computing. Our impression have been that the MPI library is more complex. Mapping MICA component model constructs onto PVM appeared straightforward, and we did not see any reason to choose a library that appears more complex for an implementation that is to serve as proof of concept. An MPI based component platform can be implemented as future work.

Implementation of EMs

Use of the `pvmd` daemon is the source of the host abstraction in PVM. The host abstraction maps nicely to the EM construct in the MICA component model. While this provides a straightforward implementation for a component platform implementation for clusters of single processor machines, which has been our main focus for this implementation, the problem of multiprocessor

machines needs more work. Running multiple `pvmd` daemons on a single host, which is possible but not straightforward in PVM, may provide the solution for supporting multiprocessor machines.

EM creation and deletion is currently not supported. Instead, all hosts that will participate in the virtual machine, all of which will become an EM, are added using the command line tool called `pvm`, which is used in PVM for launching and monitoring hosts and tasks. The RTI-PVM is then started as a task under PVM using `pvm`. The CUIs in the system can get the list of IDs of EMs present in the system using the `Get EM List` service. This choice is not based on impossibility of implementing EM creation and deletion in RTI-PVM, since PVM allows tasks running in the virtual machine to create and delete hosts, in which case the PVM uses the `rexec` service to run `pvmd` remotely. Rather, it was chosen to take this approach because this implementation allows us to test the applicability of the component model using minimum development time.

Implementation of EUs

In PVM, one of the `pvmd` daemon's tasks is to spawn new processes. The executable of RTI-PVM is the process we use as the process being spawned. Such a process serves as the EU controller for an EU in the system, and hosts the components by dynamically loading them.

The structure of this EU controller is an extension of our single threaded component platform implementation, RTI-st. The extensions include communicating service requests and messages with other EUs, a different instance registry management mechanism, and a new EU registry.

Implementation of Identifiers

PVM provides unique host and task ids in the virtual machine, which we have used in the implementation of EM and EU identifiers for RTI-PVM.

The ID of an instance is assigned by the EU controller in which the instance is created. It is built using the task ID, which also contains the host ID, and an instance specific part locally created by the EU controller. Therefore, guaranteeing the uniqueness of instance IDs reduces to simply ensuring that the local instance IDs given by an EU controller are unique among the instances in that EU.

Implementation of instance IDs in this way influences how the instance creation service works when creating component instances in EUs other than the EU that hosts the CUI which invoked the instance creation service. Since the instance ID of the new instance is to be learned from the remote EU, the instance creation service cannot return to the calling CUI CB until instance creation is complete at the remote EU. In a distributed environment, this is relatively costly. Furthermore, this cost does not incur only for the CUI that requested the instance creation service. All other component instances that share the same EU with that CUI will also be blocked, since component instances in an EU are multiprogrammed. This effect should be taken into account when deciding how the components of a system are to be distributed onto multiple EUs. A similar blocking method is also used for creation of links, where the service needs to be able to report as error when the link with the given link ID is already is in use.

Implementation of Message Passing

PVM provides non-blocking, reliable message sending, and blocking or non-blocking message reception with messages buffered at the receiver.

In RTI-PVM, message passing involves serialization and deserialization of the messages sent between instances in different EUs. The serialization and deserialization code is considered as part of the implementation of the message, therefore they are packed into a DSO (dynamic shared object). This allows us to confine the implementation of a function working on a specific message type into one piece of shared code, instead of having implementations of these functions compiled into various DSOs that include implementations of components that make use of this type of messages. These functions are used by the RTI-PVM, along with the registered message-types for links and knowledge about the links setup, to determine when a message should be serialized for transfer, to serialize it, and to deserialize and construct the proper message object from the serial form in order to put it into the receivers message queue.

When messages are sent between two instance in the same EU, the messages are never copied nor serialized/deserialized, but passed simply as pointers.

Invoking the Platform

A system built on RTI-PVM is started differently than the RTI-st. The easiest way to start the system is to do it through the `pvm` program. When `pvm` is started, it starts the local `pvmd` daemon, and waits for further commands. Then, using the `add` command, the hosts can be added to the virtual machine. These hosts will appear as EMs in the RTI-PVM. Finally, the RTI-PVM executable is run using the `pvm`'s `spawn` command. As in RTI-st, the RTI-PVM should be provided with the CU component ID from which the first instance will be created, and the message type ID for the first message to be sent to this first instance. The command line options for the RTI-PVM are the same as the options supported for RTI-st, which were given in Table 4.7.

The distributed execution of the RTI-PVM can be stopped by using the `pvm`'s `kill` command with any of the tasks running on the virtual machine. The task that is killed communicates the shutdown status to all the other tasks that belong to the same RTI-PVM execution, which then exit together with the killed task.

Not-Implemented Services and Further Improvements

The services that are not currently implemented in the RTI-PVM are replacement of the CB of a UMI, creation and deletion of EMs, and deletion of EUs. Implementation of all these services are straightforward, given time. Furthermore, they are not critical for our goal of building proof of purpose simulators on RTI-PVM, since EMs are created by using the `pvm` tool, deletion of EMs and EUs are not necessary for simulators with static run-time structure, and the replacement of the CB of a UMI is not a service normally employed in the CUI of a simulator.

4.5.4 Implementing a System on RTI-st and RTI-PVM

Implementing a system on the RTI-st and the RTI-PVM involves implementing messages and components. In this section, the steps to be followed by the developers are described in more detail.

Implementing Messages

The relevant classes in the C++ API that are used for implementing messages are the `Message` and the `MessageType` pure abstract classes. The purpose of the `MessageType` class was presented

previously in Section 4.5.2.

While there is a type system defined in the MICA component model for messages, there is also a need for a parallel object-oriented type system for the messages in the C++ environment. Therefore, all messages are implemented as subclasses of the `Message` abstract class in the C++ API. This interface ensures that all message implementations provide the methods for querying the type of the message, serialization, and deserialization.

In order to implement a new message type, the developer starts by creating a C++ header and a source file. In the header file, a new class that inherits from the `Message` abstract class in the C++ API is declared. This header is to be used by the developers of the components that send or receive messages of this type, in compiling the DSOs for the components. The implementation of the declared class goes into the source file, where the following points need to be observed:

- A function that carries the string form of the message type ID as its name should be defined with C linkage, using the modifier `extern "C"`. This function should take no arguments, and return a pointer to a new object created from the message class being implemented, as a pointer to the superclass `Message`.
- A global variable whose name is formed by adding `_mid` to the string form of the message type ID should be declared. This variable should be a pointer to an object of type `MessageType`.
- The method `mfGetMessageType` that is inherited from the superclass (class `Message`) should be implemented to return the pointer stored in the variable described in the previous item, as the value of its output parameter.
- The serialization and deserialization methods inherited from the superclass (class `Message`) should be defined.

In addition to these, good design requires the data fields in the message type to be declared private, and accessor/modifier methods to be used. When all implementation is done, the source file of the message is compiled to create a new DSO (dynamic shared object). A single DSO contains the implementation of a single message type.

Message implementations are loaded when component instances register the message types they are going to be using during their execution. When a DSO that includes a message implementation is loaded, the component platform creates a new object from the `MessageType` class, representing the unique type of the message, and store it into the variable mentioned above in the second item.

The function in the first item above is a factory function for messages to be created by the platform. In the case of RTI-st, this factory function is only used for the first message to be sent to the first CUI. In addition to creation of the first message, it is also used in transparent deserialization of messages an EU receives from other EUs in the distributed component platform implementation.

The `mfGetMessageType` addresses the need for finding the type of a message object. Although various alternatives exists for implementing this functionality, our solution takes only one method pointer per message class, only one message type object per message class, and costs only a simple function call. Message type checking is a frequent operation in the component-based systems implemented using the MICA component model.

Implementing Components

Implementation of components from a developers point of view is similar to implementing messages. However, unlike the case in implementing messages, the component implementers are not required to provide any header files. All there is to a component implementation lies in a single DSO, and a single DSO includes a single component.

For a component, the following should be addressed in its source files:

- A global function that is named the same as the string form of the component identifier should be declared with no parameters, and with the return type of `UmiCbAmbassador` or `CuiCbAmbassador` depending on whether the component is a UM or CU, respectively. This function should be declared to be `extern "C"`, which prevents its symbol from appearing in mangled form in the final DSO created for the component.
- Implementation of a class, which we may call a customized behavior (CB) class, that implements one of the interfaces mentioned above, and provides the behavior of the component.

The rules of conduct to which the objects created from a CB class should adhere, can most easily be descried by looking at the methods of a CB class and their intended function:

Constructor: The constructor of a CB object is expected to initialize the object's member variables, and other objects in its implementation if there are any. The component platform can not be used yet, since the object does not know how to access the services.

mf<Instance Type>BaseCreated(): This function is inherited from the ambassador being subtyped. Depending on whether the instance being created is an instance of an UM or a CU component, the function is named `mfCuiBaseCreated`, or `mfUmiBaseCreated`. The platform implementation calls this method in order to inform the instance about the address of the base object it is associated with, from which it will obtain services from. Typically, this method is where the CB object will register the message types that it will be using.

mfReceiveMessage(): This method, which is also inherited from the ambassador being subtyped, is called by the platform for informing about an incoming message. Whenever this method is called, the instance is expected to process the message, free the resources allocated for the message object, and return to the platform. This is where the behavior of the component is implemented. There is a single form of this method for the CB classes of UM components since they receive messages only from links. In contrast, there are two overloaded forms for the CB classes of CU components: one for receiving from links, and one for receiving messages sent using CUI IDs.

Destructor: The destructor should clean up the object's variables. While the base is still operational when the destructor is called, shutting down the CB of an instance by sending it a shutdown message before deleting it would be a cleaner design, instead of doing some cleanup using services from the platform in the destructor of the CB object, such as deleting other instances.

When a CUI requests the creation of a new instance, our component platform implementations check whether the DSO that includes the implementation of the component is already loaded. If

not, the platform requests loading of the DSO from the file whose name is obtained by adding ".so" to the component ID. Then it searches for the component ID as a symbol in the loaded DSO, and creates the CB for the new instance. When the CB object is created, it is coupled with a new instance base, and the CB is informed about the ambassador of the base it is associated with, completing the creation of the component instance. In the current versions of our component platforms, loaded DSOs are not unloaded, even if all instances of the component in the DSO are deleted from the run-time structure of the system. This is not critical to testing the component model, therefore it is left as future work.

More on Using The Same Components Under Both RTI-st and RTI-PVM

Since RTI-st and RTI-PVM support different control flow constructs, this section addresses how and to what extent the same components can be used under both platform implementations.

With respect to message sending and reception, control flow is fixed from the point of view of a single component: it receives a message, runs until it completes processing of the message, and hands the control back to the platform implementation by returning from the message reception callback function. In addition, a component instance is given only limited guarantees about message sending, as was described in Section 4.3.2 where we introduced messages. Since UM components are only capable of message sending or reception, their implementation is independent of the control flow constructs supported by any component platform. Naturally, for a UM component that will be used under different component platforms, it should be taken into account that all the requirements in the contracts of the component should continue to be satisfied. Such requirements may include statements that restrict placement of the instances created from the UM component in certain configurations with respect to EUs, EMs, and other components (i.e. being a singleton in an EU). The same argument about message sending applies in the case of CU components as well.

However, message sending and reception are not the only services provided to CU components. Through EM, EU, and component instance creation services, a set of CU instances in a system decide the following:

- A partitioning of the set of all component instances in the system. Component instances from different sets S_i and S_j of this partition run in a parallel manner. Each such set is associated with an EM in the system.
- Partitions of each of the sets S_i in the previous item, into $S_{i,k}$, so that the component instances in different sets $S_{i,m}$ and $S_{i,n}$ will be run in a preemptively multitasked manner. The component instances in one of the sets $S_{i,k}$ will process their messages by taking turns, and running their processing to its completion.

It is likely that component instance distribution as described above will preferably be different under the single threaded and the distributed platforms. Therefore, while CU components can run without any changes under both of our platforms due to the fact that the APIs remain the same, their implementation should also be able to adapt to changes in the availability of support for various control flow constructs in the component platform.

It should also be noted that while the control flow stays the same when observed from the point of view of a single component, the control flow of the whole system may change when

different platforms are used, or even across different runs on the same platform if there is some non-determinism involved in the platform implementation.

Chapter 5

DINEMO

DINEMO, whose name actually comes from a slightly distorted form of Distributed NEMAN, is the result of refactoring and extending the NEMAN emulator for mobile ad-hoc networks [149, 150].

The main objective of our work on DINEMO is to observe usability of the MICA component model in development. Therefore, DINEMO serves as a proof of concept demonstration for MICA. In addition, our work also has the following objectives that are from the point of view that DINEMO is a network emulator:

Reusing experience: NEMAN is developed within the research group that the author has been working in. Therefore considerable experience on the emulation approach employed in NEMAN existed within the research group. We wanted to organize this experience in loosely-coupled components which can be recomposed into different combinations. This will allow to fast prototype ideas on emulators of different designs and capabilities for various experiments.

Distributing NEMAN: As will be presented in Section 5.1, all the computational load of a network is put onto one single PC in NEMAN. Although this has the advantage of decreasing the costs of experimenting, it also causes limited resources to be available for the programs running on virtual nodes, or for the computing intensive tasks such as realistic physical layer simulation. Refactoring on MICA provides an implementation of the emulation approach in NEMAN that is easily distributable.

Alternative models and extensions: The component model of MICA provides ease of replacement of model implementations used in the emulation approach in NEMAN.

The design of DINEMO was published in [74]. The material in this section is an extended and updated form of what was presented in that paper.

5.1 NEMAN

NEMAN (Network Emulator for Mobile Ad-Hoc Networks) [149, 150] is a network emulator that aims to provide a low cost, scalable, and portable experimentation platform for mobile ad-hoc network protocol and middleware development. NEMAN aims to require little initial effort in

terms of installation and learning. While being kept simple, NEMAN can simulate networks with many nodes on a single PC using the TUN/TAP universal network driver.

NEMAN's architecture is divided into three collaborating parts: user processes, topology manager, and the graphical user interface (GUI) (see Figure 5.1). The topology manager and the GUI run in the user privilege level on Linux. However, superuser privileges are needed for configuring the TAP virtual network interfaces, and either processes need to be given the capability to use the `SO_BINDTODEVICE` option, or they should be run with superuser privileges.

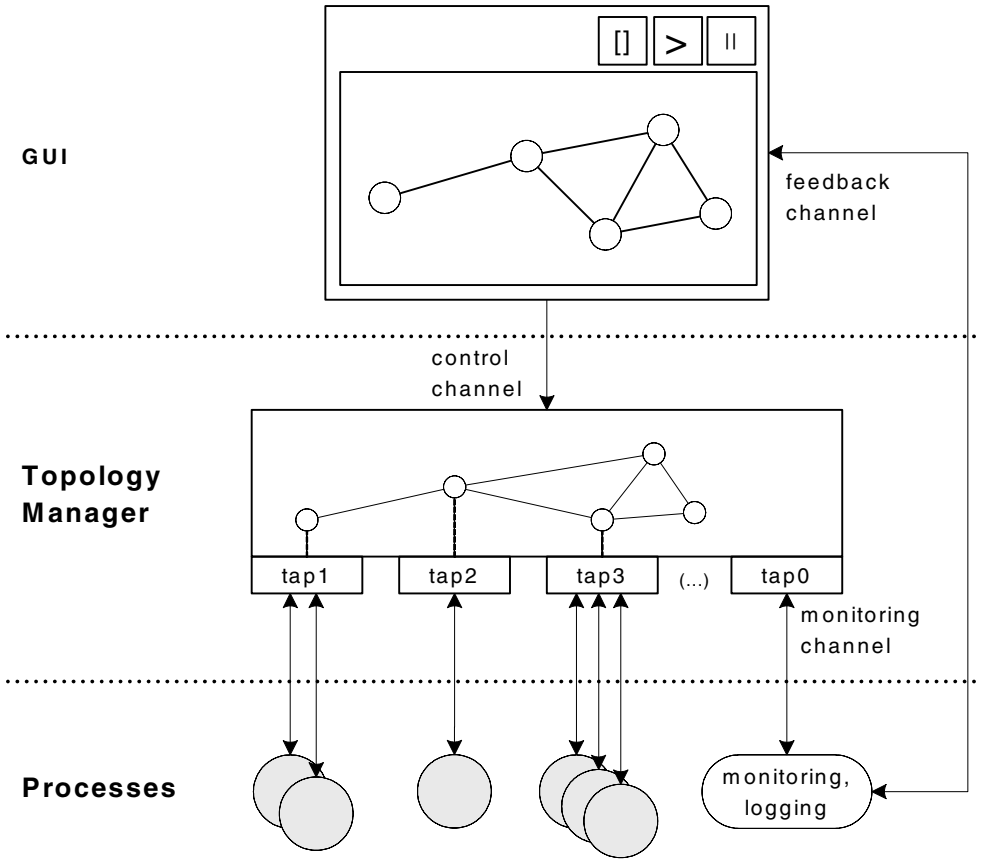

Figure 5.1: Architecture of NEMAN, taken from [150].

The applications that will be run on the network emulated by NEMAN hook to the virtual network devices. A TAP interface acts as a network interface, but it is connected to an associated file descriptor as its other end. Any frame passed to the TAP interface by the network protocol stack in the kernel appears as Ethernet frame data on the associated file descriptor, and any data written as an Ethernet frame is made available to the operating system network protocol stack through the TAP interface. Normally, Linux kernel ignores frames coming to one of its interfaces from another one of its interfaces, therefore a small patch called "send-to-self patch" is applied.

The programs choose the specific network interface they will use by making use of the option `SO_BINDTODEVICE` in the socket API. A group of processes, including for example a routing

daemon, hooked to a device in this manner represent the applications running on a virtual node in the simulated network.

NEMAN's network simulation is limited to the physical, data link, and the routing part of the network layer. More detailed physical layer simulation is a relatively new addition to NEMAN, which will not be discussed in this section.

5.1.1 Simulating Link-Level Connectivity

The link-level simulation is handled via collaboration between the GUI and the topology manager. Every frame that is written to any of the TAP interfaces is read by the topology manager, and switched between the TAP interfaces according to the connectivity information that is in effect at the moment. The connectivity information is fed to the topology manager by the GUI through a control socket. The GUI reads this information from an NS-2 style TCL script. One TAP interface, tap0, is used as a monitoring channel where all the traffic in the whole network can be monitored using a packet sniffer.

5.1.2 Simulating Routing

The routing part of the network layer is also simulated by NEMAN, as routing protocols would not normally set routes for the addresses of interfaces on the same kernel, which happens to be the case in NEMAN. Therefore, NEMAN bypasses the kernel routing tables, and uses the olsrd.org OLSR routing protocol daemon hooked to the TAP interfaces. The topology manager then simulates hop-by-hop packet forwarding by sending a copy of the packet with properly updated MAC address to every node that the packet should have traversed before reaching its destination.

The reader with some knowledge of routing will notice that this behavior of routing in NEMAN is not exactly the same with what happens on real networks. A small example (taken from [74]) would help demonstrating how packets are handled in NEMAN.

Suppose we have four nodes in a mobile network, and call them N_1, N_2, N_3, and N_4. Further assume that they are in such a configuration that N_1 can communicate with N_2, but not with N_3 or N_4, N_2 with N_1 and N_3 but not N_4, N_3 with N_2 and N_4 but not N_1, and N_4 with only N_3.

What happens in ARP protocol and routing normally in such a mobile network when N_1 tries to communicate with N_4 is shown in Figure 5.2. First, N_1 notices that it does not have the MAC address of the next hop when sending a packet to N_4, and broadcasts at layer 2 an ARP request asking for the MAC address of N_4 or the next hop to N_4 (Fig. 5.2, 1). N_2 receives the request and replies with its own MAC address (2). Receiving the reply, N_1 sends a packet destined to N_4's IP address, in a frame destined to N_2's MAC address (3). N_2 replaces the MAC address in the frame with the next hop's, N_3's MAC address and send forward the frame (4). N_3 does the same, and the packet arrives at N_4, its destination at layer 3 (5).

When the same communication happens under NEMAN where different TAP interfaces running on one Linux kernel acting as the nodes, a slightly different sequence of events happen, as shown in Figure 5.3. Again it starts with N_1 noticing that it does not have the MAC address of the next hop, therefore it sends an ARP request (Fig. 5.3, 1). The ARP request is intercepted by the topology manager in NEMAN (2), and it replies by constructing an ARP request on behalf of the next hop (3,4), but it returns the MAC address of destination N_4, instead of N_2, which is the next hop. Receiving the ARP reply, N_1 sends its packet with destination as N_4's IP, encapsulated in a

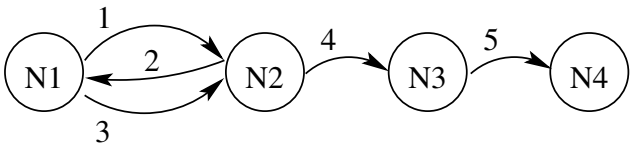

Figure 5.2: Normal operation of the ARP protocol.

frame with destination as N_4's MAC (5). This frame is also intercepted by NEMAN (6). NEMAN consults the global routing table it has constructed from the outputs of routing daemons running on all nodes, finds out which nodes lie on the route between N_1 and N_4, and sends a copy of the frame to each node on the route (N_2, N_3, and N_4), properly updating the MAC addresses of the frames sent to appear as if the packet has been following the route between N_1 and N_4 (7).

The reason for the necessity of handling routing in NEMAN is that all TAP interfaces share the same kernel routing table. It is not possible, up to the best of the author's knowledge, to make in a simple way multiple routing protocols running on a single kernel to update this shared table in such a way so that kernel would route a packet arriving to one of its interfaces from another one of its interfaces, to the next hop which is also one of its interfaces.

On the other hand, the reason for handling ARP in NEMAN is that kernel replies from all its interfaces in reply to an ARP request that comes to one of its own interfaces, which results in hundreds of ARP replies on a host that has hundreds of virtual nodes on it. Therefore passing ARP requests back to the kernel results in unnecessary traffic.

While the way hop-by-hop routing and ARP work in NEMAN differs from the protocol described in RFC 826, most of the programs on the nodes are still able to work as if they are a part of a mobile ad-hoc network.

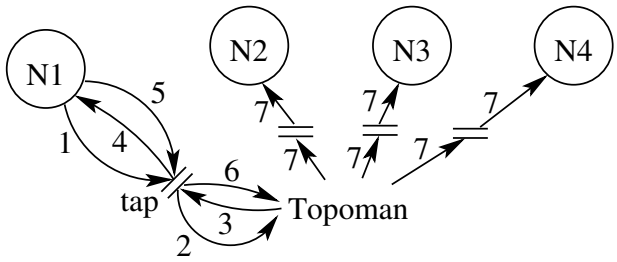

Figure 5.3: Operation of the ARP protocol and routing under NEMAN.

5.2 Initial Design of DINEMO

Our initial design for the componentization of NEMAN involved four UM components:

SelectHub-UM: A UM component whose instances act as a hub in an EU for managing the `select()` system function call, in order to keep track of the activity on file descriptors on behalf of the UMIs in the EU. As `select()` can be used as a blocking call, the call should be made from only one component in an EU.

Tap-UM: A controller that creates/destroys a TAP interface using the TUN/TAP kernel driver, and manages the data flowing through the TAP interface by collaborating with the SelectHub-UMI in the EU it is created in.

ArpEmu-UM: This UM component was planned to reply to the ARP requests coming from the Tap-UMIs in an EM, to circumvent the problem with the ARP implementation in the Linux kernel mentioned in Section 5.1.2.

Topoman-UM: Topoman-UM component was planned to be the equivalent of topology manager in NEMAN. It would provide the layer 1 and 2 simulation, hop-by-hop routing, as well as communication with the GUI.

These UM components were planned to be created and destroyed by a CUI. One example of planned run-time structure of components is shown in Figure 5.4. We have implemented the Tap-UM, SelectHub-UM, and ArpEmu-UM, along with a UM component that simulates a broadcast layer 1 and 2 communications as if every Tap-UMI is a node in a network on a broadcast medium, and demonstrated with a CU component that constructs the run-time on a single EU. When we started looking into distributing the TAP-UMIs on multiple EUs on different EMs, we have discovered a problem with the design of the ARP and routing emulation.

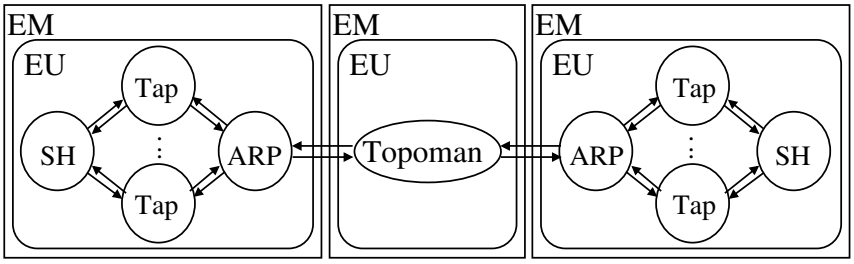

Figure 5.4: An example configuration of the simulator according to our initial design for DINEMO. (SH: SelectHub, ARP: ArpEmu)

5.2.1 Problems with ARP and Routing in Distributing NEMAN

As described in Section 5.1.2, ARP protocol and routing are treated in a special way in NEMAN. The ARP protocol has to be handled in a distributed version of NEMAN as well, since still the goal is to multiplex multiple virtual nodes using multiple virtual network interfaces on the computers running the distributed emulation. We have explored different alternatives for handling ARP in such a distributed emulation, such as having an UMI for each node for handling the ARP requests

issued by that node, having one ARP handler UMI for each computer for handling the ARP requests among the nodes on that computer, or having a central ARP handler as part of Topoman-UMI. The preferred solution for the problem of how we should componentize, was defined as the largest coherent component that can be placed as close to the source of the requests as possible.

Since the ARP requests are destined to broadcast MAC addresses, it turns out that any of the solutions above would necessitate the full list of IP-MAC address pairs for all nodes in the simulated environment, because the ARP handler either would have to be able to look up the MAC address being inquired using the given IP and reply to all ARP requests, or the frame has to be broadcasted at layer 2. If the frame is to be broadcasted, the frame should travel down the networking stack and back to all the TAP interfaces that fall into the broadcast range of the node from which the request was originated. As the kernel would reply on all its interfaces to an ARP request coming from one of its own interfaces, in order to prevent hundreds of messages being created, the ARP handler has to intercept ARP requests arriving from the topology manager, and return a reply telling its IP.

5.3 Final Design of DINEMO

The solution involving broadcasting described in the previous section about handling the ARP protocol in distributing the NEMAN, is almost the ARP protocol as described in RFC 826. Therefore we decided to explore further division of nodes into components, and distribute the topology manager in NEMAN as much as possible.

For this purpose, we started looking into using *TUN* interfaces instead of TAPs. The difference between TUN and TAP interfaces is that TAP interfaces intercept frames at layer 2, while TUN interfaces intercept packets at layer 3. This also means that the code in the kernel about routing and ARP handling is in effect when using TAPs, whereas TUN interfaces intercept IP packets before they get to these parts of the kernel.

Keeping kernel's involvement limited can be considered both as an advantage and a disadvantage. The disadvantage stems from the fact that additional code for simulated equivalents of routing, ARP, and data-link layer code in kernel are needed, in addition to physical layer simulation. However, as this provides greater control over the code simulating these layers, the simulator becomes more configurable and extensible.

The design for DINEMO includes the following UM components:

Tun-UM: The Tun-UM is the counterpart of Tap-UM described in Section 5.2. It creates/destroys a TUN virtual network interface, and manages the data flow.

Router-UM: Router-UM components simulate the routing functionality in the network layer, possibly by interacting with a routing daemon they invoke and manage, or by receiving commands from other components about next-hops to other nodes or networks. This second method is used for compatibility with NEMAN, since NEMAN receives route updates from a control socket listening to a certain port.

ARP-UM: ARP-UM component simulates RFC 826 compliant ARP protocol handling for a single node. It is capable of all ARP functionality, therefore it can both initiate ARP requests and produce replies, and it has its own ARP cache.

DLL-UM: DLL-UM components implement different data-link layer protocols. Designing DLL functionality as a UMI provides us with the means to replace the DLL implementation without changes to other instances.

PhySim-UM: PhySim-UM components implement physical layer simulation. Instances created from the PhySim-UM components can be expected to be the most computation intensive component instances in simulators. Therefore PhySim-UM components can further be divided into a set of components whose instances will be distributed over multiple computing resources. For compatibility with NEMAN, a PhySim-UM component has been developed that is capable of receiving the physical connection status from the GUI through the DINEMO's CU component.

SelectHub-UM: The SelectHub-UM component, which was implemented for our initial design and described in section 5.2, was reused here.

In addition to these UMIs, a DINEMO-CU component is developed. This CU component reads a configuration file, and creates the run-time structure accordingly. During the execution of an episode, it handles the communication through the control socket, for example with the GUI. The commands received through the control socket are dispatched to the relevant UMIs.

The design of DINEMO is in essence a component-based framework. The component implementations that have been developed are exemplary, and can be replaced with other simulator components that adhere to the same contracts. Details of the contracts of these components are given in Appendix F.

An example configuration for this TUN based approach is given in Figure 5.5. The Figure also shows how some of the UM components developed earlier for the TAP-based version can be reused in combination with the TUN-based simulator UMIs. For the case where there is at most a single TAP virtual interface in the kernel of any host used in the experimental setup, the combination is fairly easy: a DLL-UMI is connected between a Tap-UMI and the PhySim-UMI. In this case, the kernel's ARP and routing routines would work as intended. However, although it was reported as possible to put multiple TAP interfaces onto a single host using an ARP handler UM component in [74], we have discovered new problems with this approach that needs more work to be solved.

5.3.1 Running DINEMO

DINEMO is invoked by giving the DINEMO-CU as the first CU component to be instantiated by the component platform, and sending it a start message. The DINEMO-CUI reads a configuration file named `dinemo.cnf`.

In the first line of this configuration file three integers are expected: the total number n of nodes to be created, the number m of EMs in the list that follows in the configuration file, and the index (starting from 1) of the EM in the list on which the PhySim-UMI will be created, which is a singleton in the system in the current design of DINEMO. Then, the list of EMs are given in the following m lines, each line containing a computational resource ID, which is the string form of the EM ID, and the number of nodes to be created on the EM.

The run-time structure is created by the DINEMO-CUI according to the information contained in this configuration file. In the current version, the execution is finished by sending a terminate signal to one of the EMs in the component platform.

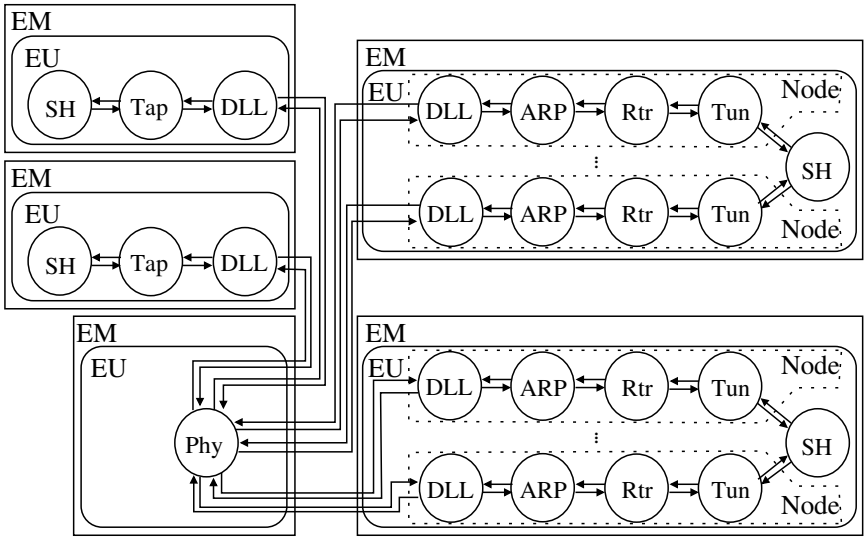

Figure 5.5: An example configuration of a DINEMO simulator, along with some Tap-UMIs reused in separate EMs. (SH: SelectHub, Rtr: Router, Phy: Physical Layer Simulator)

Chapter 6

Comparison to Related Work

6.1 MICA and Other Component Architectures

In this section, a side-by-side comparison of MICA with other component-based approaches presented in Appendix C and ACA is presented. While some of these architectures are widely used, it should be noted that except for ACA, these approaches are not targeted towards network simulators, or simulators in general. A comparative summary is provided in Table 6.1.

Differently from Appendix C where the architectures are presented one by one, the presentation below provides focused summaries organized along the following dimensions: control flow, lifetime management, component communication, and composition style. In addition, a summary of MICA's features along these dimensions is also presented. It must be noted that MICA provides a component model, and implementations of component platforms for any of the architectures presented in Appendix C may be considered for use in implementing component platforms for MICA. Such platform implementations are possible as long as the constructs in the MICA component model can be meaningfully mapped to constructs in these architectures.

6.1.1 Control Flow

How control flow is established among different components or component instances is an important issue that has influence on performance and manageability of code, since badly managed concurrency can become a source of hard to find defects.

Different architectures have different approaches to control flow:

XPCOM: Mozilla XPCOM [176] focuses on systems where the component instances share a single application thread. Therefore, it does not directly specify control flow. In the application thread, XPCOM relies on the observer pattern to deliver the Gecko (the environment) generated events, upon which component instances are activated. In addition, component instances may invoke methods in other component instances' interfaces directly.

Fractal: Fractal Component Model [25] does not specify how control flow is managed. Control flow related issues are considered to be a problem to be solved by the implementers of the abstract concept of "component membranes" that encapsulate components in the run-time.

Table 6.1: Comparative summary of MICA and other component models.

Component Model	**OMG's CORBA CCM**	**JavaBeans**	**EJB**	**COM**	**MICA**
Component types	•Basic •Extended	Bean	•Stateless, stateful session •Entity •Message driven (stateless)	COM Class	•Unit Model •Constructor Unit
Interface types	•Facets – Recepticles •Event Sources – Sinks •Attributes	•Java interfaces •Listener pattern using Java reflection •Attributes	A bit complex. Roughly: •Java interfaces •CORBA (attributes)	Immutable set of methods	Links, bound to addresses local to components
Composition style	Context-based	By wiring	Context-based	By wiring	By wiring
Control flow	•Serialize •Multithread	Orthogonal	Serialize. Reentrancy is optional (discouraged).	Apartment model, virtual threads	•cooperative multitasking, •time-shared multitasking, •and multiprocessing
Communication methods	•Function calls •Events	•Function calls •Events (*-cast)	•Function calls •Messages	•Function calls •Asynchronous calls (since W2K)	•Messages (asynchronous)
Distribution support	CORBA	RMI	RMI, RMI over IIOP, CORBA	Transparent	Transparent
Language dependency	APIs	Java	Java	None (binary)	APIs (currently only C++)
Component Platform	Various platforms, by various vendors	Java RTE	Java RTE + container implementations	Windows XYZ, Macintosh, Unix	Various platforms possible. Currently: RTI-st, RTI-PVM They use DSOs.

UNO: Universal Network Objects (UNO) [16, 26, 159] control flow model is expressed using different interface method invocation styles: direct, spawning, asynchronous, or synchronous. Direct calls are blocking. On the other hand, spawning calls are non-blocking and thread creating. Asynchronous calls are non-blocking and sequentially ordered, with each non-blocking call also thread creating. Synchronous calls are blocking, but their return is synchronized to end of all previous non-blocking sequentially ordered calls. Remote invocation options are limited to blocking and one-way non-blocking styles.

CCA: Since blocking, procedure-call like control flow maps almost directly to the virtual method table based method invocation mechanism in object-oriented languages, it is considered to be very efficient for component instances that are hosted in the same process. As Common Component Architecture (CCA) [17] declares performance as being in their focus, communication among components in the same process is established in this way. For components in different processes or different hosts, various services are expected to be provided using separate interfaces that are to be implemented by the component platforms (frameworks in CCA terminology).

ACA: In the Autonomous Component Architecture (ACA) (see Section 2.7.4), each message received by a component is processed by creating a separate execution context. There is also support for blocking style method calls, which blocks the caller until the message sent is processed by the receiver.

CCM: OMG's CORBA Component Model (CCM) [135] specifies two threading models: "serialize" and "multithread". In the serialize model, the container of the component protects the

component from multiple simultaneous accesses. In contrast, the component is responsible for taking care of such simultaneous invocations all by itself in the multithread model.

JavaBeans: Concurrency is considered to be orthogonal to the JavaBeans. Control flow is left to be designed by the developers using various Java libraries.

EJB: The context of an Enterprise JavaBeans (EJB) bean serializes all invocations to the bean's methods. Re-entrancy is allowed only when an EJB bean is explicitly declared to be re-entrant.

COM: The Microsoft's Component Object Model (COM) uses what is called the apartment model. In this model, COM objects are placed in two types of apartments: single threaded apartments (STA), and multiple threaded apartments (MTA). An STA provides a single thread for one or more COM objects, and there can be multiple STAs per process. In contrast, an MTA allows multiple threads for multiple COM objects, but only a single MTA is allowed per process.

.NET: .NET does not provide a component model, but it provides some constructs using which component platforms for various component models can be developed. Therefore .NET does not specify control flow issues for component models.

In the MICA, the model of control flow for all components is uniform: a component instance is activated only on reception of a message, and runs to completion for processing the message when activated. When a message will be passed on to a component instance, is in the discretion of the component platform the component instance is running on.

The control flow model may be fixed from a single component's perspective, but concurrency can be managed at the level of composition. MICA provides two container entities: Execution Units (EU) and Execution Managers (EM). These provide three different models of concurrency: multiprogramming (a.k.a. co-operative multitasking), multitasking (a.k.a. time-shared multitasking), and parallel processing. The component instances in an EU are multi-programmed: at an instant only one component instance is allowed to process a message. Component instances in different EUs in a single EM are multi-tasked, and execute their messages in a time-shared manner. Component instances in EUs in different EMs process their messages in parallel.

Fixing the control flow from a component's perspective provides an easy model for the component developer, and delegates the resource management responsibility to the composer of the system.

6.1.2 Lifetime Management

Lifetime management concerns how the component instances are created and deleted. In different architectures covered in Appendix C, two major methods for component creation can be observed: functional and declarative. In the functional methods, again two methods are apparent: usual object generation, such as in JavaBeans, or use of a library that provides functions for creation and deletion of component instances, such as in XPCOM, COM, .NET (if one considers assembly loading as component creation), CCA, Fractal, and UNO. CCM also has support for a functional approach though the use of the component home in a factory design pattern. The declarative approach is

used with container-based architectures, such as CCM and EJB, which allow declarative attributes being associated with components. In this approach, various lifetime policies define the lifetime of a component, based on some communication and persistence abstractions such as session, message, or entity. For component deletion, either a function is provided, or collective garbage collection with reference counting on the interfaces is used, or it is left to the pre-determined policies invoked through some declared attributes.

Among the different architectures presented, ACA is the only one that specifies neither how component creation nor deletion is to be done, leaving such issues to the component platform.

MICA follows the functional approach and provides component creation and deletion through a method. However, differently from all the architectures above, MICA provides component creation and deletion functionality only to components of the Constructor Unit (CU) type. This way, separation between model building and simulator construction is reinforced. Confinement at the architectural level of instance creation and deletion to instances of a certain component type is unique to MICA.

6.1.3 Component Communication

In this section, we will discuss how component instances find out about destination component instances to communicate to, and what communication mechanisms are available in various component models. It should be noted that finding out about destination component instances is also related to lifetime issues (creation/deletion of communication paths) and composition method.

XPCOM: XPCOM provides an event mechanism, in addition to direct invocation of interface methods. Events are sent to named topics (channels). In order to receive events, a component instance must declare an nsIObserver interface, and register this interface using a global function using the name of the topic. Therefore, in order to communicate, components have to agree off-line on a set of topic names.

In order to discover interfaces when using direct invocation of interface methods, a global function is used to find the service manager, which in turn is used to query the required interface by interface name, contract name, or interface ID.

CCM: CCM also supports both method invocations, and events. The facets are the interfaces through which methods of a component can be invoked. There exists a mapping of facets to CORBA object interfaces using a fixed naming scheme. Therefore clients can discover and invoke methods through CORBA mechanisms implemented by the ORB. If an object reference from a component is available, a client can also use the CORBA::Object::get_component method in order to convert the object reference to a reference to the component that handles the object. The client can then navigate the available interfaces through the component's equivalent interface.

Event sources in CCM can be connected in a 1-1 (emitters) or 1-n manner (publishers). Receivers of events (event sinks) cannot distinguish or choose between event sources, without help of additional mechanisms.

JavaBeans: JavaBeans beans can communicate by producing and consuming events, as well as invoking methods directly in order to modify properties, which may in turn produce "change

of value" events. Event distribution uses the listener pattern, in which event listener beans register a listener object at the event source, and the event source calls the registered event listeners' methods in the case of the event happening. JavaBeans does not provide any method to distribute information about availability of event producers or other beans. Such a mechanism has to be coded into the system developed or various libraries should be employed. This approach appears more like constituting a framework than being a part of a component model level approach.

EJB: How EJB beans communicate, and how they find out about other EJB beans, are determined by the services they require from their containers. Therefore the communication is left out from the model, to be implemented through use of various libraries (packages).

COM: COM uses invocation of methods in interfaces. Different forms of communication might also be used through various libraries available on the Windows platform.

.NET: .NET has direct support for events based on the observer pattern, on the level of objects. How objects find out references to other objects is not fixed, but it is possible to locate assemblies, which include classes, through a registry. Other than this object-level event support, various libraries available on the Windows platform can also be used.

CCA: For in-process communication, CCA components use direct invocation of methods in interfaces (ports). Various services that are expected to be made available by the component platform (framework in CCA terminology), are used for invoking methods of remote components.

A component uses the services object it is associated with, in order to declare the name of interfaces (ports) it needs, and to obtain or release references to these interfaces.

Fractal: Fractal components use direct invocation of interface methods. The BindingController interface allows a component instance to expose methods for connecting to other component instances' "server" interfaces. Introspection can be used to obtain names of interfaces a component provides. Distribution of knowledge about existence of components is left to be determined by the component platform or implemented in the system being developed.

ACA: ACA component instances send messages only through the component platform. Names of component instances and their ports are represented in a tree structure according to parent/child relationships among component instances. Root of this tree is the component platform itself. Component instances can navigate this tree in order to find other component instances. A component would have to know how the components are named (a naming scheme), and how the components in the application are organized, in order to find a component port they will send a message to. How exactly the message is sent is left to be determined by the component platform implementation.

UNO: UNO uses direct invocation of interface methods. A component uses the service manager to obtain a service, which is a set of interfaces, then uses introspection to find out about the interfaces in the service.

Communication in MICA is done using 1-1 links between incoming and outgoing links of component instances. A component is built using assumptions about what type of data is expected from an incoming link, and when. A component developer should thoroughly document such assumptions, which would form part of the contract between the instances of the component, and other component instances. Such a contract would also specify what type of data is produced and when, on the outgoing links of a component instance.

The messages are mediated by the component platforms for MICA. Furthermore, messages are not immediately passed to the receiver, but queued for reception. Therefore, MICA's message passing resembles 1-1 event producer-consumer implementations, using a single event reception method that is expected to distinguish between different events using the extra information (the link ID) provided by the platform in addition to the message. COM provide similar interfaces, called delegate interfaces, that has a single method that logically distinguishes method invocations using additional information provided.

Different from various component models covered, MICA provides only this unicast event-like mechanism for components that are to be used for implementing the application logic (Unit Model Instances – UMIs). This is because MICA is designed to be a minimalistic, and multicast and broadcast communications can be provided by additional components that would form a new architectural tier implemented using MICA.

A component instance that is sending a message uses the address of the outgoing link, which is local to the sending component. Receiving component instances can distinguish the incoming link a message was received from, based on the address of the incoming link, which local to the receiving component instance.

For construction of links, the identifiers of the component instances are needed, which are pieces of information with system-wide semantics. Unconstrained use of such information with system-wide semantics has the potential to make components in the systems tightly coupled, thus such information is confined to instances of Constructor Unit (CU) components in MICA. Since CU component instances (CUIs) are allowed to make use of such IDs, they are allowed to use CUI ids to communicate with other CUIs as well. Instance IDs cannot be used for communication with the UMIs, since they have no proper way of receiving such messages. Such confinement of link creation and deletion to the instances of a particular component type, which are expected to be used in less numbers in applications, is again unique to MICA.

6.1.4 Composition

Methods of composition seem to have a correlation with the domains of application: the component architectures that target the enterprise computing market, namely CCM, EJB, and .NET, supports mainly context-based composition. The form of composition for the other architectures covered in this thesis, namely XPCOM, JavaBeans, COM, Fractal, ACA, and UNO, is composition by wiring. In fact, Fractal takes a middle position: the concept of membranes that serve as containers for composite components provide the means to implement context-based composition, while components are also composed by wiring their gates together, including the composite components.

MICA follows a very strict form of composition by wiring. When a component instance receives a message through an incoming link, it receives the message and the link identifier through a

single callback function. Similarly, when sending a message, it uses a single function in its BaseAmbassador, providing it with the link identifier of the outgoing link to send the message to. Messages sent to links that are not connected are silently dropped, so that a component instance may continue working even when all its links are not connected. Of course, it might not have been anticipated by the developers that an instance of their component will find itself working in an environment where the receiver of one or more of its links are missing. Such a situation would most probably point to an error in construction or management of the run-time structure of the system.

In addition, and differently from all other architectures, the ability to form connections between incoming links and outgoing links of component instances is provided only to instances of Constructor Unit components in MICA.

6.2 MICA and Other Component-Based Approaches to Network Simulation

After the comparison presented in the last section between MICA and various component models, in this section we will compare MICA to the component-based approaches to network simulation, which have been reviewed in Section 2.7, namely TeD, OMNeT++, and OSA. Comparison with ACA have already been covered in the previous section.

It is easy to point out the similarities between MICA and these approaches: they have a notion of component, and the composition approach involves composition by wiring. Like MICA, TeD and OMNeT++ also explicitly take the position of attempting to align components of the SUT and the model components, as defined in Section 2.4.

It must also be noted that when the differences are analyzed, it appears that providing an implementation of the approaches in TeD and OSA using MICA should be possible. This is also partly true for OMNeT++: the fact that OMNeT++ allows direct method invocations between components, cannot be easily mapped on to MICA.

6.2.1 MICA and TeD

MICA and TeD differ in the following aspects:

Control flow: MICA's control flow approach was described previously in Section 6.1.1. In TeD, all behaviors in an entity have an associated thread. Considering that TeD's composite entities with their containment style composition can be implemented using multiple components in MICA, with an ambassador component logically encapsulating the contained components, MICA allows more control flow options to the simulator developers in terms of control flow.

Component packaging: This is actually an issue about the component platforms. In the component platforms implemented for MICA, the components and messages are packed as dynamically loaded shared libraries (DSOs), which exist and compiled independently from the component platform. In TeD, a component exists as a set of C++ source files and a TeD file, which are compiled using a special compiler into a simulator.

Setup of run-time: In MICA, setup of the run-time, that is the creation of components and their wiring, is accomplished through instances of components of a special kind, the Constructor

Unit components. In TeD, the setup functionality is built into the simulation library that is compiled into the simulators. This library creates instances of entities and connects them according to the descriptions provided in a configuration file.

Component platform and SMF: In MICA, the component platforms exist as separate executables, and the simulation management functionality (SMF) is implemented as sets of components. A MICA component can be used with multiple different MICA component platforms, provided that they share the APIs, as exemplified by our two implementations presented in this thesis. In TeD, the component platform and SMF exists as a library. They are used through a special compiler, and compiled into the simulator.

Composite components: MICA does not allow composite components, while TeD allows composite components in containment style. As discussed in Section 2.3.6, such composite components can be mapped to sets of components in MICA.

6.2.2 MICA and OMNeT++

The differences between OMNeT++ and MICA are summarized below. How most of the following issues are addressed in MICA was presented in the previous section when discussing MICA and TeD, therefore we will present only the OMNeT++'s approaches here, and point to differences when they are not immediately apparent.

Control flow: OMNeT++ provides two control flow options to modules, which is the name used for components in OMNeT++: components can run uninterrupted until they finish processing an incoming message, or they are implemented as a co-routine. The run-to-completion scheme is similar to MICA component design, however OMNeT++ lacks the additional control flow options available to the simulator developers in MICA. The co-routine based model requires a separate stack per component. It appears possible to implement co-routine based execution on MICA components, without any need to change MICA's design. However, this is not addressed in this thesis.

Component packaging: Like TeD, components exist in OMNeT++ also as a set of C++ files, which are compiled into the simulator.

Setup of run-time: The simulation library uses a configuration file written using the NED language, which is specially designed for this purpose, in order to create the instances of the components (modules) compiled into the simulator, and their wiring in the run-time.

Component platform and SMF: Both the component platform and the simulation management functionality (SMF) exist as an object-oriented library to be compiled into the simulator, along with the available modules.

Composite components: OMNeT++ supports aggregation style composite components. Under MICA, flat forms of composite components can be used, as discussed in Section 2.3.6.

Communication between components: Communication between the component instances in MICA is strictly limited to messages sent or received using inlinks and outlinks. While

OMNeT++ defines gates and connections, which more or less correspond to links in MICA, it also allows direct method invocations between the objects in the implementation of different components (modules). This presents an obstacle to finding an easy way of expressing OMNeT++'s component model in MICA.

6.2.3 MICA and OSA

The OSA is built on the Fractal component model, therefore most of the issues with respect to which we have compared TeD and OMNeT++ to MICA have already been covered in Section 6.1. One remaining issue is how the simulation management functionality is implemented in OSA. MICA takes the position that SMF should be implemented as a set of components, which are not different in essence from those used to implement the models. In OSA, the preferred place to implement the SMF is in the component membrane. The module that implements the SMF, which may be an object, can be shared between component membranes of different components. Since Fractal allows componentized component membranes, exposing interfaces of a component that implements the SMF and that is shared between component membrane implementations of different components is also considered to be a possibility. Yet another option is to implement the SMF as a shared component.

Both the membrane based and the shared component based methods can be expressed in MICA's component model. The membrane serves as the encapsulator for a single component instance, or component instances in case of containment style composite component formation. In either case, the membrane has its own set of behaviors, which means that its status can easily be upgraded to a normal component. This naturally leads to the flat-model equivalent of the containment-style composite components, where the set of contained components are abstracted behind a logically encapsulating component. The simulator interface which is implemented in the membrane of a component in OSA, then can be mapped to such encapsulating components in the flat component model of MICA.

Shared components are components that are regarded to be contained in more than one composite component. When mapped to a flat component model, these shared components become normal components whose connections to other components do not necessarily go through the logically encapsulating components of the sets of components that from the equivalents of composite components.

Therefore, it appears that the approaches taken in OSA and MICA about implementation of SMF are not the same due to use of composites or contexts (membranes) in Fractal. However, they are not incompatible either, given a transformation from Fractal's component model which allows composites, to the the MICA's component model which is flat.

6.3 DINEMO and Other Network Emulators

The primary motivation for implementing DINEMO was demonstration of the feasibility of the MICA approach, not just implementing another network emulator. Nevertheless, we are going to compare DINEMO to other TUN/TAP virtual network interface based emulators known to us: NCTUns, and EmuNet.

A detailed comparison with NEMAN is considered unnecessary, since the ideas in DINEMO

are based on NEMAN. DINEMO has a more modular code organization with respect to NEMAN, due to its use of MICA. Some models that are available in NEMAN are yet missing in DINEMO, such as more detailed physical layer simulation, and some models in DINEMO are not available in NEMAN, such as a more accurate ARP simulation.

6.3.1 DINEMO and NCTUns

NCTUns is a simulator, which can also be used in some emulation-based experiments where the real implementation of the protocol stack in a UNIX-like operating system is needed to be used [185]. The operating system should support creation of TUN virtual network interfaces. The structure of NCTUns is very similar to NEMAN, but while NCTUns uses TUN interfaces that intercept packets between the network and data-link layers, NEMAN uses TAP interfaces that intercept frames after the data-link layer.

DINEMO also uses the TUN virtual network interfaces. There are two main differences between DINEMO and NCTUns:

DINEMO is component-based: The DINEMO is based on MICA, therefore it is component-based, and its model implementations are organized as loosely coupled components. The structure of the code of the simulator in NCTUns does not appear to be a major focus for its developers, and it is not component-based.

DINEMO supports distribution: The PVM based distributed component platform that is developed for MICA allows distribution of DINEMO in a way that is transparent to the model developers. The experimenter, through a configuration file, decides which machines will contribute to the emulation, and how many virtual nodes will be multiplexed on each. NCTUns, on the other hand, does not provide a distributed simulator[1].

6.3.2 DINEMO and EmuNET

EmuNET is another network emulator that uses the TUN virtual network interfaces [99]. An instance of the emulator is run as a process for each virtual node to be created, which reads the packets intercepted by the virtual interface, simulates the connection characteristics between the virtual node associated with it and the target virtual node of the packet, and routes them to other EmuNET processes in the same or other physical hosts. It also receives packets sent by other EmuNET processes and forwards them to the virtual TUN interface it is associated with. Therefore, an emulation-based experiment can be distributed onto multiple physical hosts using EmuNET.

The main differences between DINEMO and EmuNET are as follows:

DINEMO is component-based: As already mentioned in the previous section, DINEMO is built on MICA, and thus it is component-based. EmuNET is implemented in plain C, using multiple threads. The code structure does not appear to be of major concern.

DINEMO allows more flexible model composition: EmuNET focuses on the aggregated effects such as bit-rate limitation and packet loss, below the level of the network layer. There is no

[1] However, note that Wang et al. present the NCTUns simulator as distributed, drawing onto the fact that a remote GUI driven control program can start multiple copies of their monolithic simulator for different simulation runs.

further detailing of models in the simulator. However, in DINEMO, the layers below the network layer are implemented using a set of components that implement different protocols. This allows different protocol implementations at different detail levels to be built for DINEMO. Furthermore, the component-based code organization of DINEMO would even allow replacement of detailed protocol components with more abstract ones. The structure of model composition is very flexible in DINEMO, due to MICA.

Chapter 7

Possible Future Directions

7.1 On Deepening Understanding about the Subject Areas

There is always more work to do in understanding a subject area and its relations with related subjects. As the survey parts of this thesis show, the subject areas related this thesis are quite big, and there are a lot to understand.

Understanding component-based software technology is key to understanding some of the new technologies such as service-oriented computing, which has become very popular in the enterprise computing domain in the recent years. Section 2.3 very briefly summarizes some related issues in component-based software engineering. By extending the discussions and descriptions, Section 2.3 can be extended up to a lecture module, or a full size book on component-based software technology.

Due to its popularity, it is also worth looking into service-oriented computing more closely. Exploring the relationship between the component-based and service-oriented approaches, especially with a focus on the potential benefits and drawbacks of using service-oriented approaches for engineering simulators and emulators, is a study we wish to see the results of.

Survey in Appendix C can be extended in various ways. The most obvious way would be to find other component-based approaches and discussing their component models. An equivalently valuable way is extending the discussion of the component models along the lines of any new issues formulated as a result of understanding the component-based technology better. Furthermore, as can also be observed from our survey, a list of techniques and characteristics that can form a sound basis in order to make it possible to conduct comparative studies on component-based approaches, is yet to be formulated.

Another interesting question is whether there are enough questions, challenges, tools, and techniques that can be put together in order to pave way towards a new discipline which can be called "simulator engineering". Given a model to be simulated, a practitioner of this discipline would focus on engineering and building simulators that simulates the model effectively, and in accordance with various experiment-specific requirements.

The survey on component-based approach to simulation, which is presented in Section 2.4, can be further developed. This survey has lead to the identification of three conceptual levels in which the term "component" is applied in different ontological contexts. Unfortunately, such useful classifications can rarely be identified up front, before a survey is conducted. Therefore based on Section 2.4, further work can take this identification and apply it back onto the available literature

for a more structured survey.

The survey on network emulators and testbeds, which is presented in Appendix B, can also be extended. A good starting point would be adding the emulators already identified in Section B.4.

Further work directions regarding the problems specific to emulation presented in Section 3.5 can also be formulated. Such a study requires further surveys to precisely identify what kinds of experiments can be effected from which of the emulation specific problems, as a first step. Then, a synthesis of the common characteristics of these experiments that lead to these problems, can be used in designing and conducting new experiments that aim to measure the effects of these problems. Finally, guidelines for avoiding them should be formulated.

7.2 On MICA

7.2.1 Component Model

The component model in MICA is currently stable and usable, as demonstrated by our component platform implementations and our work on the emulator DINEMO. Nevertheless, some extensions are still possible.

It appears beneficial to take a closer look at what the contracts for components in MICA should include, and how these contracts can be documented. Whether more complex type systems for messages are necessary or useful, is another issue that is related to the contracts, and should also be addressed.

Versioning is another question that can be looked into as future work. Versioning becomes especially important in component-based systems where components serve as units of loading, which have been the case for the platform implementations of MICA. If a component A uses two other components B and C, both of which depend on two different versions of a component D, then it should be possible to load the different versions of D side-by-side into the run-time environment.

MICA component model has been developed with the goal of addressing the needs of network simulators and emulators. However, it may prove useful in other simulation domains as well, in which the run-time structure of the components in the simulated part of the stand-in for the system under test (SIFSUT) does not change once the episode is started. Exploring usefulness of MICA in domains other than network simulation and emulation also remains as future work.

Yet another subject that needs more work is formulating how systems based on MICA can interoperate with systems based on other component models. Such a work would be beneficial in paving the way towards interoperating a simulator built on MICA and a simulator built on some other component model, such as J-Sim which is built on ACA.

7.2.2 Component Platforms

Our PVM-based component platform (RTI-PVM) is the first prototype distributed component platform implemented for the MICA component model. Therefore, its implementation has provided considerable feedback for the component model, and it had to evolve in order to conform to changes being made in the model. Therefore, it currently needs a new major revision. However, since component model is stable now, the new major revision can be designed to allow optimiza-

tions. This is possibly the most "near future" work, along with adding support for creation and deletion of EMs by the CU components, deletion of EUs, and addressing the problem of an extra copy being made in the implementation of message passing because of some missing PVM features in the PVM library we have been using.

Another point for further development involves working across computers with different architectures. We have not yet tested the RTI-PVM in such a context. Although PVM works across different architectures, some minor issues related to data formats have to be checked in the RTI-PVM before it can be deemed usable across platforms with different architectures.

While we have deliberately stayed away from XML for the reasons discussed in Section 4.5.1, the XML document object model (DOM) might be explored for providing support for XML in the object based messaging scheme in our component platform implementations. The two message description methods may even exist side-by-side at run-time, using a customizable message implementation that can handle DOM. Such exploration of using DOM also remains as possible future work.

About using dynamic linking and loading, it would be beneficial to take a closer look at symbol hiding and other mechanisms, which might prove useful for supporting and enforcing isolation between component implementations in better ways. In relation to this, another interesting question is whether it is possible to implement component platforms that uses dynamic linking and loading, in a way that allows different compilers to be used for compiling different components and the platform implementation.

Lastly, many more component platforms that use different technologies can be built. Among different alternatives, an MPI based component platform, and a component platform that works on Microsoft's Windows operating systems might be interesting. A Java API along with a component platform written in Java can also be developed. To allow for components working on different component platform implementations to interoperate, an inter-platform communication protocol should be designed. Finally, high performance component platforms that are optimized for particular machine architectures such as shared memory multiprocessor machines, would be interesting to develop.

7.2.3 Tools

It is possible to define sets of tools that might come in handy when building systems using MICA. One set of tools, which can be developed as a result of further research on use of contracts in MICA, would consist of tools for contract management. Such tools would help component developers to write contracts for their components, and system builders to find out which components can be composed with which others in order to construct the system.

Other tools that can be created include tools for supporting the design and implementation phases. For example, one subject that might be further researched is support for different design-level views of the system, which in the end would be mapped onto MICA constructs. Such a tool can be used to present a design environment to the designer in which constructs that are not found in MICA, such as different flavors of composite components, can be used in designs. Tools for implementation, on the other hand, may provide support to the component developers through a development environment, for helping with the tasks needed to be carried out in creation of component and message implementations.

7.2.4 Useful Component-Based Frameworks and Libraries

In addition to supporting simulator interoperability between simulators built using MICA, MICA can also be used as an environment for interoperating simulators that are not built using MICA or for using different simulators that are not built using MICA as part of a new simulator that is being developed using MICA. In order to realize this, wrappers for different simulators should be written, which might represent the simulator in the MICA run-time environment as a single component instance, or a set of component instances that correspond to models or virtual entities in the simulator that is being wrapped.

It is possible to define various useful component-based frameworks for simulators that will be built using MICA. One such class of frameworks that we would like to address as future work is frameworks for modeling virtual time, or in other words implementing time management services. Since simulation of virtual time is not regarded as part of MICA, different approaches can be developed as component frameworks. Availability of multiple frameworks implementing different time management techniques would make it possible to study coexistence of different time management techniques in a simulator. Another related study would involve looking into whether co-existence of multi-programmed, multi-tasked, and parallel running component instances in the run-time structure of a system impedes with or facilitates the implementation of time management services.

Another useful component-based framework would be one that supports effective high bandwidth data gathering and I/O support, which would be used for implementing logging in network simulators, for example.

At the component level, building support on UM components for co-routine based component implementations would make it more easier to implement simulators for models built using the process-oriented world-view.

7.3 Simulators and Emulators

The foremost goal of our work on DINEMO was to provide a proof-of-concept example about the usability of the MICA component model in development of a network simulator. DINEMO is in essence a component-based framework, and our focus has been on making a simulator design using the MICA. The components developed for DINEMO are examplary, and they serve as a proof-of-concept showing that simulators are implementable using this component-based framework. Therefore, while performance tests are required for any simulator that has been built, the performance of the examplary component implementations has not been our foremost concern. However, developing new and "production level" components, and conducting performance tests on such components, are among the most easily foreseeable subjects for future work.

More future work subjects can be formulated that can follow our work on DINEMO. Promoting DINEMO for use by various researchers is important, in order to be able to collect feedback on both DINEMO, and MICA. There are also some problems to be addressed in the methods employed in DINEMO, such as finding a solution that allows using multiple TAP virtual interfaces on a single host, and implementing a no-copy message passing solution for the components

implemented for DINEMO[1].

Naturally, another easily identifiable future work direction is building more simulators and emulators using MICA, that support different techniques for simulation and integration of entities used as real. As more simulators become available, it would also become possible to formulate more questions and identify more problems about simulator interoperation, model replacement, and model reuse.

[1]The problem mentioned here is not about message passing in the component platform, which is no-copy if communicating instances are on the same EU. This problem is about processing network packets in the component instances in an episode, which implement protocols or layers.

Chapter 8

Concluding Remarks

At the end of this thesis, it would be most proper to go back, and pay a second visit to the question that has driven the studies behind it: "what can be better ways of realizing network simulators and emulators?" As pointed out at the start of the thesis, a definite and complete answer to this question is a blurry and moving target.

In seeking partial answers to this question, we first broke it down into its parts, and attempted to find out about what the question really asked. In doing so, we had to formulate our own stances on simulation, emulation, and testbed based experimentation, and we had to present our view on a selected set of issues in component-based software engineering to serve as anchor points for the rest of the study. A lesson learned is that such anchor points rarely come as early in the preparation of the thesis as one would hope for. Therefore, understanding is a constant turmoil in formulating a template to fit to the variety of works in a subject area, and breaking down and rebuilding your template as works or concepts that do not fit in the template become so numerous that one can no longer ignore them as outstanders. In fact, contrasting this to the theory presented in one of the major works in the philosophy of science, the theory of paradigms in Thomas Kuhn's "The Structure of Scientific Revolutions" [105], this constant turmoil should be regarded as nothing but constantly building and replacing personal paradigms. Chapters 2 and 3, Appendices B and C, and partly the Chapter 6 are products of this process.

The background study in this thesis has been intentionally conducted as extensively as possible. Although this has made the text of this thesis relatively long, we consider that the background study in this thesis constitutes a major part of the thesis. The reason is that component-based development of network simulators and emulators involves multiple disciplines in computer science and engineering. A firm understanding in each subject involved, and construction of relationships between the different motivations, goals, attitudes, and approaches takes a considerable part of any multi-disciplinary study.

In relation to the background studies, we would like to make a remark about the survey in Appendix C. Although we are not the first to introduce the terms "component model", "component platform", "component-based architecture", and "component-based framework", the systems available in the market today appear to have been designed without much regard for the separation of the concerns being referred to by these terms. Therefore, although we have named Appendix C as "Survey of Available Component Models", one can rarely find a direct description of their component models in the documentations available. Therefore, one can also regard that survey as a survey of component-based approaches, systems, and infrastructures that appear to be the most popular

ones as of the writing of this thesis, with an attempt to put special emphasis on their component models.

Our Minimalistic Component-Based Architecture (MICA) is a software architecture which aims to appear intuitive and simple, yet expressive and useful for implementing the simulators and emulators that are needed by the network researchers in conducting their experiments. Our choice of a component-based approach was motivated by the following:

- Use components in order to avoid a costly global analysis, and use coherent, self-contained units that can be analyzed separately.
- Provide independent extensibility.
- Provide the base part of a software architecture by focusing on the component model and the platforms, that should be extended by implementing functional units.

MICA provides the developers with the necessary structures to build coherent, self-contained units. It provides support for separate analysis of components though keeping the coupling between the component low. In addition, our component platforms help by providing an environment where the components are packed separately.

In our component model and the component platforms that support it, the connection topology among the component instances can be modified at run-time. From such modification, the instances of worker components do not get affected at all. However, instances of constructor components, which manage the run-time structure, can be sensitive to the changes in the run-time connection topology. Therefore, MICA cannot provide independent extensibility by its own, but does support it if the constructor components in a system are also implemented with independent extensibility in mind.

With regard to the third motivation, MICA demonstrates that the component model and component platform concerns can be separated from component-based architecture, component-based framework, and application concerns. Applications can be built on MICA by providing a constructor component whose instance creates and composes instances of worker and constructor components available to the component platform.

With regard to the preferred characteristics for the component model we seek, presented in Section 4.3.1, MICA stands as follows:

- Striving for completeness: In a MICA-based system, the system is composed of the component platform and a set of components. No object oriented or procedural code is needed for startup. Run-time structure, which is composed of component instances and communication links between them, is also constructed and managed by instances of a special type of components, called Constructor Unit components. However, due to non-component-based nature of contemporary operating systems, component instances may still need to get services by invoking procedural APIs, which prevents MICA-based systems from completely satisfying the completeness property.
- Supporting components with variable granularity: MICA provides the minimum requirement for both small and large components, by providing asynchronous message sending, and component instantiation on remote hosts. In addition, message passing is 1-1, therefore it does not necessitate any complex channel management or data routing.

- Being simple and minimal: MICA aims to be simple. However, simplicity is relative, and a large user base is needed to measure the observed simplicity of the architecture. We aimed to ensure simplicity through minimizing the architecture, and the rules guiding our design decisions towards a minimal architecture were presented in Section 4.3.1. MICA includes 13 constructs in the component model. The APIs provide 16 services to constructor component instances, and only 2 services to worker component instances. Compared to the functionality it provides, and taking into account that control flow options are also explicit in MICA component model, we believe that these numbers are relatively small.

- Separating worker and run-time management components: In MICA, Unit Model type components correspond to worker components, and Constructor Unit components correspond to constructor components.

- Supporting transparent distribution of components: MICA supports distribution of components, in a way that is transparent to the worker components. Constructor components cannot be made independent of the distribution of component instances. The reason is that they are provided with services for managing the configuration of the component instances at run-time, which necessitates that they can have and may use information about the distribution of component instances.

- Using messaging for component communication, and preferring asynchronous messaging over synchronous: MICA uses messages for communication, and provides asynchronous messaging between component instances.

As discussed in more detail in Sections 6.1 and 6.2, the design decisions involved in MICA do not fully overlap with any major component-based approach in software engineering, or any of the component-based approaches to simulation that has been surveyed. Therefore, MICA is presented as the only component model that is designed with particular emphasis on the set of characteristics mentioned above.

One of the goals in this thesis have been a thorough presentation of the design choices in formulation of the MICA component model and implementation of the component platforms. In retrospect, it had been one of the major sources of the author's confusion and surprise throughout the development process of both the component model and the component platforms, that despite MICA defines a minimalistic component model with a very small number of entities, still a considerable number of design decisions had to be made. The discussions provided in this thesis can be used by other future attempts in building other component-based models for simulation and emulation, and even for other domains. In time, such design decision documents would enable researchers to construct the design space for component-based infrastructures for simulators and emulators, and to identify the fundamental and supplementary sets of techniques that span this space.

It should also be noted that after seeing the amount of design decisions involved, we have become quite intrigued that the answer might be positive to the question of whether there are enough questions and challenges for establishing a sub-discipline of software engineering, which one might call "simulator engineering".

The network emulator DINEMO has been just a demonstrative tool for the purposes of this thesis. Therefore, it is still in its infancy. In time, it may or may not become widely used. That it

becomes widely used is not only dependent on its sound component-based architecture that is built on the MICA component model, but more so on the models available for it, the demand for the TUN/TAP based emulation technique used in its implementation, and its proper promotion and support in the networking research community. Although one may claim that the benefits of MICA cannot be observed before a simulator or an emulator, such as DINEMO, becomes widely used, it should be taken into account that such promotion can not easily be fitted into the limited time available for a doctoral thesis. This is especially so for this thesis since the research areas that form the background of this thesis are quite large, inadequately organized, and they had to be surveyed separately which was quite time consuming.

Bibliography

[1] Ahn, G., and Chun, W. Design and implementation of MPLS network simulator (MNS) supporting QoS. In *Proceedings of the 15th International Conference on Information Networking* (2001), pp. 694–699.

[2] Ahn, J. S., and Danzig, P. B. Packet network simulation: speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking 4*, 5 (1996), 743–757.

[3] Ahn, J. S., Danzig, P. B., Liu, Z., and Yan, L. Evaluation of TCP Vegas: emulation and experiment. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '95)* (New York, NY, USA, 1995), ACM Press, pp. 185–195.

[4] Akkøk, N. *Towards the principles of designing diagrammatic modeling languages: some visual, cognitive, and foundational aspects*. PhD thesis, University of Oslo, 2004.

[5] Allen, R., Douence, R., and Garlan, D. Specifying dynamism in software architectures. In *Proceedings of the Foundations of Component-Based Systems Workshop (FoCBS)* (Zurich, Switzerland, September 1997), G. T. Leavens and M. Sitaraman, Eds.

[6] Allman, M., Caldwell, A., and Ostermann, S. ONE: The Ohio network emulator. Technical Report TR-19972, School of Electrical Engineering and Computer Science, Ohio University, August 1997.

[7] Andersen, D. G., Balakrishnan, H., Kaashoek, M. F., and Morris, R. Experience with an evolving overlay network testbed. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 13–19.

[8] Andersson, J. Reactive dynamic architectures. In *ISAW '98: Proceedings of the third international workshop on Software architecture* (New York, NY, USA, 1998), ACM Press, pp. 1–4.

[9] August, D. I., Malik, S., Peh, L.-S., and Pai, V. Achieving structural and composable modeling of complex systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (April 2004).

[10] Barr, R. *An efficient, unifying approach to simulation using virtual machines*. PhD thesis, Cornell University, May 2004.

[11] Barr, R., Haas, Z. J., and van Renesse, R. JiST: Embedding simulation time into a virtual machine. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation* (2004).

[12] Bartholet, R. G., Brogan, D. C., and Paul F. Reynolds, J. The computational complexity of component selection in simulation reuse. In *Proceedings of the 37th Winter Simulation Conference (WSC '05)* (2005), pp. 2472–2481.

[13] Bassé, M., Huiskamp, W., and Stroosma, O. A component architecture for federate development. In *Proceedings of 1999 Fall Simulation Interoperability Workshop* (1999).

[14] Bavier, A., Feamster, N., Huang, M., Peterson, L., and Rexford, J. In vini veritas: realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2006), ACM Press, pp. 3–14.

[15] Benjamin, P., Delen, D., Mayer, R., and O'Brien, T. A model-based approach for component simulation development. In *Proceedings of the 32nd Winter Simulation Conference (WSC 2000)* (December 2000), vol. 2, pp. 1831–1839.

[16] Bergmann, S. UNO execution model, October 2004. Available on-line: http://udk.openoffice.org/common/man/execution.html.

[17] Bernholdt, D. E., Allan, B. A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T. L., Damevski, K., Elwasif, W. R., Epperly, T. G. W., Govindaraju, M., Katz, D. S., Kohl, J. A., Krishnan, M., Kumfert, G., Larson, J. W., Lefantzi, S., Lewis, M. J., Malony, A. D., Mclnnes, L. C., Nieplocha, J., Norris, B., Parker, S. G., Ray, J., Shende, S., Windus, T. L., and Zhou, S. A component architecture for high-performance scientific computing. *Int. J. High Perform. Comput. Appl. 20*, 2 (2006), 163–202.

[18] Bhatt, S., Fujimoto, R., Ogielski, A., and Perumalla, K. Parallel simulation techniques for large-scale networks. *IEEE Communications Magazine 36*, 8 (1998), 42–47.

[19] Bless, R., and Doll, M. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNet++. In *Proceedings of the 36th Winter Simulation Conference (WSC '04)* (2004), pp. 1556–1561.

[20] Booch, G. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin-Cummings, 1987.

[21] Bradford, R., Simmonds, R., and Unger, B. Packet reading for network emulation. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (August 2001), pp. 15–18.

[22] Brakmo, L. S., and Peterson, L. L. Experiences with network simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 1996), ACM Press, pp. 80–90.

[23] Breslau, L., Estrin, D., Fall, K., Floyd, S., Heidemann, J., Helmy, A., Huang, P., McCanne, S., Varadhan, K., Xu, Y., and Yu, H. Advances in network simulation. *IEEE Computer 33*, 5 (May 2000), 59–67. Expanded version available as USC TR 99-702b at http://www.isi.edu/~johnh/PAPERS/Bajaj99a.html.

[24] Briot, J.-P., and Meurisse, T. An experience in using components to construct and compose agent behaviors for agent-based simulations. In *Proceedings of the AI, Simulation and Planning in High Autonomy Systems (AIS), and Conceptual Modeling and Simulation (CMS) Conference (AIS-CMS 2007) (co-located with the International Modeling and Simulation Multiconference (IMSM 2007))* (February 2007), F. Barros, C. Frydman, N. Giambiasi, and B. Zeigler, Eds., SCS, pp. 207–212.

[25] Bruneton, E., Coupaye, T., and Stefani, J. B. *The Fractal Component Model*, 2.0-3 draft ed. The Object Web Consortium, February 2004.

[26] Budischewski, J. A guide to language-independent UNO, November 2004. Available on-line: http://udk.openoffice.org/common/man/concept/unointro.html.

[27] Bunge, M. A. *Ontology I: The Furniture of the World*, vol. 3 of *Treatise on Basic Philosophy*. Springer, 1977.

[28] Bunge, M. A. *Ontology II: A World of Systems*, vol. 4 of *Treatise on Basic Philosophy*. Springer, 1979.

[29] Buss, A. Component based simulation modeling with Simkit. In *Proceedings of the 34th Winter Simulation Conference (WSC 2002)* (December 2002), vol. 1, pp. 243–249.

[30] Buss, A. H. Component-based simulation modeling. In *Proceedings of the 2000 Winter Simulation Conference* (2000).

[31] Byun, T. SIMMT-II: implementation of network simulator for IP multicast using multiple MCSs on the ATM networks. In *Proceedings of the 8th International Conference on Parallel and Distributed Systems (ICPADS 2001)* (2001), pp. 220–225.

[32] Carnahan, J. C., P. F. Reynolds, J., and Brogan, D. C. Simulation-specific characteristics and software reuse. In *Proceedings of the 37th Winter Simulation Conference (WSC 2005)* (December 2005).

[33] Carreras, I., Grasso, R., Kiraly, C., Pera, S., Woesner, H., Ye, Y., and Szabó, C. A. Design considerations on the CREATE-NET testbed. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for Development of Networks and Communities (TRIDENTCOM'05)* (2005).

[34] Carson, M., and Santay, D. NIST Net: a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 111–126.

[35] Cavin, D., Sasson, Y., and Schiper, A. On the accuracy of MANET simulators. In *POMC '02: Proceedings of the second ACM international workshop on Principles of mobile computing* (2002), ACM Press, pp. 38–43.

[36] Chambers, B. A. The grid roofnet: a rooftop ad hoc wireless network. MIT Master's thesis, June 2002.

[37] Chen, G., and Szymanski, B. K. Component-oriented simulation architecture: Toward interoperability and interchangeabiliy. In *Proceedings of the 2001 Winter Simulation Conference* (2001).

[38] Chen, G., and Szymanski, B. K. COST: A component-oriented discrete event simulator. In *Proceedings of the 34th Winter Simulation Conference (WSC '02)* (2002), pp. 776–782.

[39] Chen, W.-S. E., Lin, C.-H., and Chen, Y. C. Design and implementation of an object-oriented atm network simulator. In *Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96)* (February 1996), pp. 123–127.

[40] Chow, A. C. H., and Zeigler, B. P. Parallel DEVS: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Winter Simulation Conference (WSC'94)* (San Diego, CA, USA, 1994), SCS, pp. 716–722.

[41] Clements, N. S., Heck, B. S., and Vachrsevanos, G. Component based modeling and fault tolerant control of complex systems. In *Proceedings of the 19th Digital Avionics Systems Conferences (DASC 2000)* (2000).

[42] Cox, B. J. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[43] Dalle, O. OSA: An open component-based architecture for discrete-event simulation. In *Proceedings of the 20th Eutopean Conference on Modelling and Simulation (ECMS'06)* (May 2006).

[44] Dalle, O. OSA: An open component-based architecture for discrete-event simulation. Research Report 5762–version 2, Institutt National De Recherche en Informatique et Automatique (INRIA), February 2006.

[45] Dalle, O. Component-based discrete event simulation using the fractal component model. In *Proceedings of the AI, Simulation and Planning in High Autonomy Systems (AIS), and Conceptual Modeling and Simulation (CMS) Conference (AIS-CMS 2007) (co-located with the International Modeling and Simulation Multiconference (IMSM 2007))* (February 2007), F. Barros, C. Frydman, N. Giambiasi, and B. Zeigler, Eds., SCS, pp. 213–218.

[46] Dalle, O., and Weiner, G. An open issue on applying sharing modeling patterns in devs. In *Proceedings of the 2007 Summer Computer Simulation Conference* (July 2007).

[47] De, P., Raniwala, A., Sharma, S., and Chiueh, T.-C. Design considerations for a multihop wireless network testbed. *IEEE Communications Magazine 43*, 10 (October 2005), 102–109.

[48] Denis, A., Perez, C., Prol, T., and Ribes, A. Padico: a component-based software infrastructure for grid computing. In *Proceedings of the International Parallel and Distributed Processing Symposium* (April 2003).

[49] Diaz-Calderon, A., Paredis, C. J. J., and Khosla, P. K. Organization and selection of reconfigurable models. In *Proceedings of the 2000 Winter Simulation Conference* (2000).

[50] Doner, J. R. GENESIM: Generic network simulator. *IEEE Journal on Selected Areas in Communications 6*, 1 (January 1988), 172–179.

[51] Drepper, U. How to write shared libraries. In *Proceedings of the UKUUG Linux Developers' Conference* (July 2002). Available on-line: http://people.redhat.com/drepper/dsohowto.pdf.

[52] D'Souza, D. F., and Wills, A. C. *Objects, Components, and Frameworks with UML: the Catalysis Approach*. Addison-Wesley, 1999.

[53] Earnshaw, R. W., Titchmarsh, A., and Mars, P. Design and implementation of a packet-switched network simulator. In *Proceedings of the 6th United Kingdom Teletraffic Symposium* (May 1989).

[54] Edelstein, D., and Edwards, D. A component-based design for a simulated network. In *Proceedings of the IEEE Conference on Information Reuse and Integration (IRI 2004)* (November 2004), pp. 283–289.

[55] Ely, D., Savage, S., and Wetherall, D. Alpine: A user-level infrastructure for network protocol development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems* (2001), pp. 171–184.

[56] Engel, M., Smith, M., Hanemann, S., and Freisleben, B. Wireless ad-hoc network emulation using microkernel-based virtual linux systems. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation* (2004), pp. 198–203.

[57] Fall, K. Network emulation in the Vint/NS simulator. In *Proceedings of the The Fourth IEEE Symposium on Computers and Communications (ISCC '99)* (Washington, DC, USA, 1999), IEEE Computer Society, p. 244.

[58] Finin, T., Labrou, Y., and Mayfield, J. KQML as an agent communication language. In *Software Agents*, J. Bradshaw, Ed. AAAI/MIT Press, 1997.

[59] Floridi, L. Information. In *The Blackwell Guide to the Philosophy of Computing and Information*, L. Floridi, Ed. Blackwell Publishing Ltd., 2004, ch. 4, pp. 40–61.

[60] Flynn, J., Tewari, H., and O'Mahony, D. Jemu: A real time emulation system for mobile ad hoc networks. In *Proceedings of the First Joint IEI/IEE Symposium on Telecommunication Systems Research* (November 2001).

[61] Fox, M. R., Brogan, D. C., and Paul F. Reynolds, J. Approximating component selection. In *Proceedings of the 36th Winter Simulation Conference (WSC '04)* (2004), pp. 429–434.

[62] Frey, H., Görgen, D., Lehnert, J. K., and Sturm, P. A java-based uniform workbench for simulating and executing distributed mobile applications. In *Proceedings of FIDJI 2003 International Workshop on Scientific Engineering of Distributed Java Applications* (2003).

[63] Fujimoto, R. M. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, 2000.

[64] Fujimoto, R. M., Perumalla, K., Park, A., Wu, H., Ammar, M. H., and Riley, G. F. Large-scale network simulation: how big? how fast? In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS 2003)* (October 2003), pp. 116–123.

[65] Fukunari, M., Chi, Y.-L., and Wolfe, P. M. JavaBean-based simulation with a decision making bean. In *Proceedings of the 30th Winter Simulation Conference (WSC 1998)* (December 1998), vol. 2, pp. 1699–1702.

[66] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[67] Ganu, S., Kremo, H., Howard, R., and Seskar, I. Addressing repeatability in wireless experiments using ORBIT testbed. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for Development of Networks and Communities (TRIDENTCOM'05)* (2005).

[68] Geist, A., Baguelin, A., Dougarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[69] Göktürk, E. Emulating ad hoc networks: differences from simulations and emulation specific problems. In *New Trends in Computer Networks*, vol. 1 of *Advances in Computer Science and Engineering: Reports*. Imperial College Press, October 2005.

[70] Göktürk, E. Design of a higher level architecture for network simulators. In *Proceedings of the 20th European Conference on Modelling and Simulation (ECMS 2006)* (May 2006), W. Borutzky, A. Orsoni, and R. Zobel, Eds., SCS, pp. 232–237.

[71] Göktürk, E. Elements and stakeholders of network simulation. Tech. Rep. 338, University of Oslo, Department of Informatics, April 2006.

[72] Göktürk, E. Towards simulator interoperability and model replacability in network simulation and emulation through AMINES-HLA. In *Proceedings of the 38th Winter Simulation Conference (WSC 2006)* (December 2006), L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, Eds., SCS, pp. 2170–2179.

[73] Göktürk, E. A stance on emulation and testbeds, and a survey of network emulators and testbeds. In *Proceedings of the 21st European Conference on Modelling and Simulation (ECMS 2007)* (June 2007), SCS. (to appear).

[74] Göktürk, E., Pužar, M., and Akkøk, M. N. Distributing NEMAN network emulator using MICA component architecture. In *Proceedings of the AI, Simulation and Planning in High Autonomy Systems (AIS), and Conceptual Modeling and Simulation (CMS) Conference (AIS-CMS 2007) (co-located with the International Modeling and Simulation Multiconference (IMSM 2007))* (February 2007), F. Barros, C. Frydman, N. Giambiasi, and B. Zeigler, Eds., SCS, pp. 199–205.

[75] Golmie, N., Mouveaux, F., Hester, L., Saintillan, Y., Koenig, A., and Su, D. *The NIST ATM/HFC Network Simulator: Operation and Programming Guide*. U.S. Department of Commerce, NIST Information Technology Laboratory, Advanced Networks Technologies Division, Gaithersburg, MD, USA, December 1998.

[76] Goswell, C. The COM programmer's cookbook. Microsoft Developer's Network (MSDN) Library, COM General Technical Articles, 1995. Available on-line: http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_com_co.asp.

[77] Guffens, V., and Bastin, G. Running virtualized native drivers in user mode linux. In *Proceedings of USENIX 2005 Annual Technical Conference* (2005).

[78] Guruprasad, S., Ricci, R., and Lepreau, J. Integrated network experimentation using simulation and emulation. In *Proceedings of the First International Conference on Testbeds and*

Research Infrastructures for Development of Networks and Communities (TRIDENTCOM'05) (2005).

[79] Haeberlen, A., Dischinger, M., Gummadi, K. P., and Saroiu, S. Monarch: a tool to emulate transport protocol flows over the internet at large. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement* (New York, NY, USA, 2006), ACM Press, pp. 105–118.

[80] Heineman, G. T., and Councill, W. T., Eds. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[81] Herrscher, D., Maier, S., and Rothermel, K. Distributed emulation of shared media networks. In *Proceedings of the 2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2003)* (2003), pp. 226–233.

[82] Herrscher, D., and Rothermel, K. A dynamic network scenario emulation tool. In *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN '02)* (2002), pp. 262–267.

[83] Herzum, P., and Sims, O. *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, 2000.

[84] Himmelspach, J., and Uhrmacher, A. M. A component-based simulation layer for james. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS 2004)* (May 2004), pp. 115–122.

[85] Horstmann, M., and Kirtland, M. DCOM architecture. Microsoft Developer's Network (MSDN) Library, 1997. Available on-line: http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp.

[86] Huang, X. W., Sharma, R., and Keshav, S. The ENTRAPID protocol development environment. In *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99)* (March 1999), vol. 3, pp. 1107–1115.

[87] Ingham, D. B., and Parrington, G. D. Delayline: a wide-area network emulation tool. *Comput. Syst. 7*, 3 (1994), 313–332.

[88] Ionescu, B., Ionescu, M., Veres, S., Ionescu, D., Cuervo, F., and Luiken-Miller, M. A testbed and research network for next generation services over next generation networks. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for Development of Networks and Communities (TRIDENTCOM'05)* (2005).

[89] Jacobson, I. *Object-Oriented Software Engineering: A Use CaseDriven Approach*. Addison-Wesley, 1992.

[90] Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, U.S.A., April 1999.

[91] Jansen, S., and McGregor, A. Simulation with real world network stacks. In *WSC '05: Proceedings of the 37th Winter Simulation Conference* (2005), Winter Simulation Conference, pp. 2454–2463.

[92] Jiang, X., and Xu, D. vbet: a vm-based emulation testbed. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research* (New York, NY, USA, 2003), ACM Press, pp. 95–104.

[93] Judd, G., and Steenkiste, P. Repeatable and realistic wireless experimentation through physical emulation. *SIGCOMM Comput. Commun. Rev. 34*, 1 (2004), 63–68.

[94] Judd, G., and Steenkiste, P. A software architecture for physical layer wireless network emulation. In *WiNTECH '06: Proceedings of the 1st international workshop on Wireless network testbeds, experimental evaluation & characterization* (New York, NY, USA, 2006), ACM Press, pp. 2–9.

[95] Kaba, J. T., and Raichle, D. R. Testbed on a desktop: strategies and techniques to support multi-hop MANET routing protocol development. In *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing (MobiHoc '01)* (New York, NY, USA, 2001), ACM Press, pp. 164–172.

[96] Karaliopoulos, A., Tafazolli, R., and Evans, B. An ns-derived GEO satellite network simulator: features, capabilities, results. In *Proceedings of the IEEE Seminar and Exhibition on Simulation and Modelling of Satellite Systems* (2002).

[97] Karrer, R., Sabharwal, A., and Knightly, E. Enabling large-scale wireless broadband: the case for TAPs. *SIGCOMM Comput. Commun. Rev. 34*, 1 (2004), 27–32.

[98] Kavimandan, A., Lee, W., Thottan, M., Gokhale, A., and Viswanathan, R. Network simulation via hybrid system modeling: a time-stepped approach. In *Proceedings of the 14th International Conference on Computer Communications and Networks (ICCCN 2005)* (October 2005), pp. 531–536.

[99] Kayssi, A., and El-Haj-Mahmoud, A. EmuNET: a real-time network emulator. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing* (New York, NY, USA, 2004), ACM Press, pp. 357–362.

[100] Kiddle, C., Simmonds, R., and Unger, B. Improving scalability of network emulation through parallelism and abstraction. In *ANSS '05: Proceedings of the 38th annual Symposium on Simulation* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 119–129.

[101] Kiddle, C., Simmonds, R., Williamson, C., and Unger, B. Hybrid packet/fluid flow network simulation. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation (PADS '04)* (June 2003), pp. 143–152.

[102] Kock, B., Wijting, C., Kuipers, M., and Prasad, R. WIP-Sim: a novel object-oriented event-driven IP network simulator. In *Proceedings of the 54th IEEE Vehicular Technology Conference (VTC 2001)* (2001), pp. 2557–2561.

[103] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *ACM Trans. Comput. Syst. 18*, 3 (2000), 263–297.

[104] Kropff, M., Krop, T., Hollick, M., S.Mogre, P., and Steinmetz, R. A survey on real world and emulation testbeds for mobile ad hoc networks. In *Proceedings of 2nd IEEE International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM 2006)* (March 2006), IEEE.

[105] Kuhn, T. S. *The Structure of Scientific Revolutions*, second edition (enlarged) ed. University of Chicago Pres, Chicago, Illinois, 1970.

[106] Lebas, F.-X., and Usländer, T. Opera: An hpcn architecture for distributed component-based real-time simulations. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking* (1999), vol. 1593 of *Lecture Notes in Computer Science*, Springer, pp. 60–69.

[107] Lefantzi, S., Ray, J., and Najm, H. N. Using the Common Component Architecture to design high performance scientific simulation codes. In *Proceedings of the International Parallel and Distributed Processing Symposium* (April 2003).

[108] Lei, J., Yates, R., Greenstein, L., and Liu, H. Wireless link snr mapping onto an indoor testbed. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for Development of Networks and Communities (TRIDENTCOM'05)* (2005).

[109] Leino, K. R. M., and Nelson, G. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst. 24*, 5 (2002), 491–553.

[110] Levine, J. R. *Linkers and Loaders*. Morgan-Kauffman, October 1999. Manuscript available on-line: http://www.iecc.com/linker/.

[111] Li, Y., and Qian, D. A practical approach for constructing a parallel network simulator. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '2003)* (August 2003), pp. 655–659.

[112] Liljenstam, M., Liu, J., Nicol, D., Yuan, Y., Yan, G., and Grier, C. RINSE: The real-time immersive network simulation environment for network security exercises. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 119–128.

[113] Liu, X., and Chien, A. A. Traffic-based load balance for scalable network emulation. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 40.

[114] Liu, X., and Chien, A. A. Realistic large-scale online network simulation. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 31.

[115] Lundgren, H., Lundberg, D., Nielsen, J., Nordström, E., and Tschudin, C. A large-scale testbed for reproducible ad hoc protocol evaluations. In *Proceedings of IEEE Wireless Communications and Networking Conference 2002 (WCNC'02)* (March 2002).

[116] Luo, Q., Ni, L. M., He, B., Wu, H., and Xue, W. MEADOWS: modeling, emulation, and analysis of data of wireless sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks* (New York, NY, USA, 2004), ACM Press, pp. 58–67.

[117] Macker, J. P., Chao, W., and Weston, J. W. A low-cost, IP-based mobile network emulator (MNE). In *Proceedings of the Military Communications Conference (MILCOM 2003)* (October 2003).

[118] Mahadevan, P., Rodriguez, A., Becker, D., and Vahdat, A. MobiNet: a scalable emulation infrastructure for ad hoc and wireless networks. In *WiTMeMo '05: Papers presented at the 2005 workshop on Wireless traffic measurements and modeling* (Berkeley, CA, USA, 2005), USENIX Association, pp. 7–12.

[119] Mainzer, K. System: An introduction to systems science. In *The Blackwell Guide to the Philosophy of Computing and Information*, L. Floridi, Ed. Blackwell Publishing Ltd., 2004, ch. 3, pp. 28–39.

[120] Maltz, D. A., Broch, J., and Johnson, D. B. Experiences designing and building a multi-hop wireless ad hoc network testbed. Technical Report CMU-CS-99-116, School of Computer Science, Carnegie Mellon University, March 1999.

[121] Matthes, M., Biehl, H., Lauer, M., and Drobnik, O. MASSIVE: An emulation environment for mobile ad-hoc networks. In *Proceedings of the Second Annual Conference on Wireless On-demand Network Systems and Services (WONS'05)* (2005).

[122] McIlroy, M. D. Mass produced software components. In *Proceedings of the NATO Software Engineering Conference* (1968), pp. 138–155.

[123] Descriptions and workings of OLE threading models. Microsoft Support Knowledge Base, article id: 150777, December 2003. Available on-line: http://support.microsoft.com/kb/q150777/.

[124] Miller, J. A., Ge, Y., and Tao, J. Component-based simulation environments: JSIM as a case study using Java Beans. In *Proceedings of the 30th Winter Simulation Conference (WSC 1998)* (December 1998), vol. 1, pp. 373–381.

[125] Mouftah, H. T., and Sturgeon, R. P. Distributed discrete event simulation for communication networks. *IEEE Journal on Selected Areas in Communications 8*, 9 (1990), 1723–1734.

[126] Naoumov, V., and Gross, T. Simulation of large ad hoc networks. In *MSWIM '03: Proceedings of the 6th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems* (2003), ACM Press, pp. 50–57.

[127] Nicol, D. M. Utility analysis of parallel simulation. In *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation* (Washington, DC, USA, 2003), IEEE Computer Society, p. 123.

[128] Nicol, D. M., Liljenstam, M., and Liu, J. Advanced concepts in large-scale network simulation. In *Proceedings of the 37th Winter Simulation Conference (WSC '05)* (December 2005).

[129] Nierstrasz, O., and Dami, L. Component-oriented software technology. In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis, Eds. Prentice-Hall, 1996.

[130] Nordström, E., Gunningberg, P., and Lundgren, H. A testbed and methodology for experimental evaluation of wireless mobile ad hoc networks. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom 2005)* (2005).

[131] Nygaard, K., and Dahl, O.-J. The development of the SIMULA languages. 439–480.

[132] Ohsaki, H., Oscar, G., and Imase, M. Quasi-dynamic network model partition method for accelerating parallel network simulation. In *Proceedings of the 14th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS 2006)* (September 2006), pp. 255–264.

[133] OMG. Unified modeling language (UML), version 1.3, 1999.

[134] OMG. Common object request broker architecture: Core specification. Adopted specification of the Object Management Group (OMG), Inc., March 2004. Version 3.0.3.

[135] OMG. CORBA component model specification. Object Management Group (OMG) Available Specification, April 2006. Version 4.0.

[136] Orfali, R., Harkey, D., and Edwards, J. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1995.

[137] Ott, M., Seskar, I., Siraccusa, R., and Singh, M. ORBIT testbed software architecture: Supporting experiments as a service. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for Development of Networks and Communities (TRIDENTCOM'05)* (2005).

[138] Ould-Ahmed-Vall, E., Riley, G. F., Heck, B. S., and Reddy, D. Simulation of large-scale sensor networks using GTSNetS. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005)* (September 2005), pp. 211–218.

[139] Park, A., Fujimoto, R. M., and Perumalla, K. S. Conservative synchronization of large-scale network simulations. In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation* (New York, NY, USA, May 2004), ACM Press, pp. 153–161.

[140] Park, S., Savvides, A., and Srivastava, M. B. Sensorsim: a simulation framework for sensor networks. In *MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems* (New York, NY, USA, 2000), ACM Press, pp. 104–111.

[141] Pérennou, T., Couchon, E., Dairaine, L., and Diaz, M. Two-stage wireless network emulation. In *Proceedings of the Workshop on Challenges of Mobility held in conjunction with 18th IFIP World Computer Congress (WCC)* (2004), F. Boavida, E. Monteiro, and J. Orvalho, Eds., pp. 57–66.

[142] Perrone, L. F., Yuan, Y., and Nicol, D. M. Modeling and simulation best practices for wireless ad hoc networks. In *Proceedings of the 2003 Winter Simulation Conference* (2003), S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, Eds., pp. 685–693.

[143] Perumalla, K. S., Park, A., Fujimoto, R. M., and Riley, G. F. Scalable RTI-based parallel simulation of networks. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS 2003)* (June 2003), pp. 97–104.

[144] Peterson, L., Anderson, T., Culler, D., and Roscoe, T. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev. 33*, 1 (2003), 59–64.

[145] Pidd, M., Oses, N., and Brooks, R. J. Component-based simulation on the web? In *Proceedings of the 31st Winter Simulation Conference (WSC 1999)* (December 1999), vol. 2, pp. 1438–1444.

[146] Pollard, J. K. Component-based architecture for simulation of transmission systems. In *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)* (2000), pp. 363–368.

[147] Polley, J., Blazakis, D., McGee, J., Rusk, D., and Baras, J. S. ATEMU: a fine-grained sensor network simulator. In *Proceedings of the First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004)* (October 2004), pp. 145–152.

[148] Praehofer, H., Sametinger, J., and Stritzinger, A. Component frameworks — a case study. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 30)* (August 1999), pp. 148–157.

[149] Pužar, M., and Plagemann, T. NEMAN: A network emulator for mobile ad-hoc networks. In *Proceedings of the 8th International Conference on Telecommunications (ConTEL 2005)* (2005).

[150] Pužar, M., and Plagemann, T. NEMAN: A network emulator for mobile ad-hoc networks. Technical Report 321, Department of Informatics, University of Oslo, March 2005.

[151] Ran, A. Software isn't built from Lego blocks. In *Proceedings of the 1999 symposium on Software reusability (SSR '99)* (New York, NY, USA, 1999), ACM Press, pp. 164–169.

[152] Rao, D. M., and Wilsey, P. A. Dynamic component substitution in web-based simulation. In *Proceedings of the 32nd Winter Simulation Conference (WSC '00)* (San Diego, CA, USA, 2000), Society for Computer Simulation International, pp. 1840–1848.

[153] Rao, D. M., and Wilsey, P. A. Multi-resolution network simulations using dynamic component substitution. In *Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2001), pp. 142–149.

[154] Riley, G. F. Large-scale network simulations with GTNetS. In *WSC '03 Proceedings of the 35th Winter simulation conference* (2003), pp. 676–684.

[155] Riley, G. F., Ammar, M. H., Fujimoto, R. M., Xu, D., and Perumalla, K. Distributed network simulations using the dynamic simulation backplane. In *Proceedings of the 21st International Conference on Distributed Computing Systems* (April 2001), pp. 181–188.

[156] Ritter, H., Tian, M., Voigt, T., and Schiller, J. A highly flexible testbed for studies of ad-hoc network behaviour. In *Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks* (October 2003).

[157] Rizzo, L. An embedded network simulator to support network protocols' development. In *Proceedings of the 9th International Conference on Computer Performance Evaluation: Modeling Techniques and Tools*, vol. 1245 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1997, pp. 97–107.

[158] Sabino, B. Non-blocking method calls. Microsoft Developer's Network (MSDN) Library, COM General Technical Articles, 1999. Available on-line: http://msdn.microsoft.com/library/en-us/dncomg/html/nbmc.asp.

[159] Schulten, D., and Kuhnert, R. OpenOffice.org developer's guide. Available on-line: http://api.openoffice.org/DevelopersGuide/DevelopersGuide.html.

[160] Shimayoshi, T., Hori, K., Lu, J. Y., Amano, A., and Matsuda, T. A software environment for simulators suitable for complex biological analysis. In *Proceedings of the 26th Annual International Conference of the Engineering in Medicine and Biology Society (EMBC 2004)* (September 2004), vol. 2, pp. 3047–3050.

[161] Simmonds, R., Bradford, R., and Unger, B. Applying parallel discrete event simulation to network emulation. In *PADS '00: Proceedings of the fourteenth workshop on Parallel and distributed simulation* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 15–22.

[162] Simmonds, R., Williamson, C., Bradford, R., Arlitt, M., and Unger, B. Web server benchmarking using parallel wan emulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2002), ACM Press, pp. 286–287.

[163] Sinha, R., Paredis, C. J. J., and Khosla1, P. K. Behavioral model composition in simulation-based design. In *Proceedings of the 35th Annual Simulation Symposium* (Washington, DC, USA, 2002), IEEE Computer Society, p. 308.

[164] Smith, J. Understanding and using COM threading models. Microsoft Developer's Network (MSDN) Library, COM General Technical Articles, July 1998. Available on-line: http://msdn.microsoft.com/library/en-us/dncomg/html/comthreading.asp.

[165] Sobeih, A., Hou, J. C., Kung, L.-C., Li, N., Zhang, H., Chen, W.-P., Tyan, H.-Y., and Lim, H. J-sim: A simulation and emulation environment for wireless sensor networks. *IEEE Wireless Communications 13*, 4 (August 2006), 104–119.

[166] Son, Y. J., Jones, A. T., and Wysk, R. A. Automatic generation of simulation models from neutral libraries: an example. In *Proceedings of the 32th Winter Simulation Conference (WSC 2000)* (December 2000), vol. 2, pp. 1558–1567.

[167] Stanovsky, D. Virtual reality. In *The Blackwell Guide to the Philosophy of Computing and Information*, L. Floridi, Ed. Blackwell Publishing Ltd., 2004, ch. 12, pp. 167–177.

[168] Stuckenholz, A., and Osterloh, A. Safe component updates. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering* (New York, NY, USA, 2006), ACM Press, pp. 39–48.

[169] Sundresh, S., Kim, W., and Agha, G. SENS: A sensor, environment and network simulator. In *Proceedings of 37th Annual Simulation Symposium* (April 2004), pp. 221–228.

[170] Szymanski, B. K., and Chen, G. Linking spatially explicit parallel continuous and discrete models. In *Proceedings of the 32th Winter Simulation Conference (WSC 2000)* (December 2000), vol. 2, pp. 1705–1712.

[171] Szymanski, B. K., and Liu, Y. Loosely-coordinated, distributed, packet-level simulation of large-scale networks. In *WSC '03 Proceedings of the 35th Winter simulation conference* (2003), pp. 712–720.

[172] Szyperski, C. Independently extensible systems: Software engineering potential and challenge. In *Proceedings of the 9th Australasian Computer Science Conference* (1996).

[173] Szyperski, C., Gruntz, D., and Murer, S. *Component software: beyond object-oriented programming*. Pearson Education Limited, 2002.

[174] Takai, M., Bagrodia, R., Gerla, M., Daneshrad, B., Fitz, M., Srivastava, M., Belding-Royer, E., Krishnamurty, S., Molle, M., Mohapatra, P., Rao, R., Mitra, U., Shen, C.-C., and Evans, J. Scalable testbed for next-generation wireless networking technologies. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for Development of Networks and Communities (TRIDENTCOM'05)* (2005).

[175] Terzis, A., Nikoloudakis, K., Wang, L., and Zhang, L. IRLSim: a general purpose packet level network simulator. In *Proceedings of the 33rd Annual Simulation Symposium* (2000), pp. 109–120.

[176] Turner, D., and Oeschger, I. *Creating XPCOM Components*. Brownhen Publishing, 2003.

[177] Tyan, H.-Y. *Design, Realization and Evaluation of a Component-Based Software Architecture for Network Simulation*. PhD thesis, Department of Electrical Engineering, Ohio State University, 2001.

[178] Tyan, H.-Y., Sobeih, A., and Hou, J. C. Towards composable and extensible network simulation. In *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium* (2005).

[179] Unger, B. W., Gomes, F., Zhonge, X., Gbursynski, P., Ono-Tesfaye, T., Ramaswamy, S., Williamson, C., and Covington, A. A high fidelity atm traffic and network simulator. In *Proceedings of the 27th Winter simulation conference (WSC '95)* (1995), pp. 996–1003.

[180] Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., and Becker, D. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 271–284.

[181] Vaidya, N. H., Bernhard, J., Veeravalli, V. V., Kumar, P. R., and Iyer, R. K. Illinois wireless wind tunnel: a testbed for experimental evaluation of wireless networks. In *E-WIND '05: Proceeding of the 2005 ACM SIGCOMM workshop on Experimental approaches to wireless network design and analysis* (New York, NY, USA, 2005), ACM Press, pp. 64–69.

[182] Varga, A. The OMNET++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)* (June 2001).

[183] Verbraeck, A. Component-based distributed simulations: the way forward? In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS 2004)* (May 2004), pp. 141–148.

[184] Vernez, J., Ehrensberger, J., and Robert, S. Nessi: a python network simulator for fast protocol development. In *Proceedings of the 11th International Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks* (2006), pp. 67–71.

[185] Wang, S. Y., Chou, C. L., Huang, C. H., Hwang, C. C., Yang, Z. M., Chiou, C. C., and Lin, C. C. The design and implementation of the NCTUns 1.0 network simulator. *Computer Networks 42* (2003), 175–197.

[186] Wang, Y.-H., and Ho, S.-H. Implementation of a DEVS-JavaBean simulation environment. In *Proceedings of the 34th Annual Simulation Symposium* (2001), pp. 333–338.

[187] Weber, R. *Ontological Foundations of Information Systems*. Coopers & Lybrand and Accounting Association of Australia and New Zealand, 1997.

[188] White, B., Lepreau, J., and Guruprasad, S. Lowering the barrier to wireless and mobile experimentation. In *Proceedings of the First Workshop on Hot Topics in Networks (Hotnets)* (2002).

[189] White, B., Lepreau, J., and Guruprasad, S. Lowering the barrier to wireless and mobile experimentation. *SIGCOMM Comput. Commun. Rev. 33*, 1 (2003), 47–52.

[190] Wiederhold, G., Wegner, P., and Ceri, S. Toward megaprogramming. *Communications of the ACM 35*, 11 (1992), 89–99.

[191] Williams, S., and Kindel, C. The component object model: A technical overview. *Dr. Dobb's Journal* (December 1994). Available on-line: http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp.

[192] Xiao, Z., and Unger, B. Synchronization and concurrent control in communication network simulation. In *Proceedings of the International Conference on Communication Technology (WCC–ICCT 2000)* (2000), vol. 2, pp. 1275–1281.

[193] Xu, D., Riley, G. F., Ammar, M. H., and Fujimoto, R. Split protocol stack network simulations using the dynamic simulation backplane. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 158–165.

[194] Yan, G., and Eidenbenz, S. Sluggish calendar queues for network simulation. In *Proceedings of the 14th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOT 2006)* (September 2006), pp. 127–136.

[195] Yeom, I., and Reddy, A. N. ENDE: An end-to-end network delay emulator tool for multimedia protocol development. *Multimedia Tools Appl. 14*, 3 (2001), 269–296.

[196] Zec, M., and Mikuc, M. Real-time IP network simulation at gigabit data rates. In *Proceedings of the 7th International Conference on Telecommunications (ConTEL 2003)* (2003), pp. 235–242.

[197] Zeigler, B. P. *Theory of Modeling and Simulation*. Wiley, 1976.

[198] Zeng, X., Bagrodia, R., and Gerla, M. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 154–161.

[199] Zhang, Y., and Li, W. An integrated environment for testing mobile ad-hoc networks. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing* (New York, NY, USA, 2002), ACM Press, pp. 104–111.

[200] Zheng, P., and Ni, L. M. EMPOWER: A scalable framework for network emulation. In *Proceedings of the International Conference on Parallel Processing 2002* (2002), pp. 185–192.

[201] Zheng, P., and Ni, L. M. EMWIN: emulating a mobile wireless network using a wired network. In *WOWMOM '02: Proceedings of the 5th ACM international workshop on wireless mobile multimedia* (New York, NY, USA, 2002), ACM Press, pp. 64–71.

Appendix A

Critique of Szyperski's No-Observable-State Characteristic

Szyperski maintains having "no (externally) observable state" as one of the characteristics of components ([173], pp. 36). This no-observable-state characteristic appear to address several things at once. Below we present a critique of this characteristic, drawing upon the definitions provided in Section 2.3.2.

Szyperski's motivation in supporting the no-observable-state characteristic appears to be related with supporting another proposition: a component should not be distinguishable from copies of its own. Any state that is not related to a component's functionality can be considered an exception, such as the serial number of the component. From this proposition, he deduces that it does not make sense to have multiple copies of a component in the same operating system process since these would be indistinguishable, and provides the example of a database server with a database. He states that although the database server and the database might be seen as a module with state, the static database server program is a component, and the database is a supported instance — a database "object". Although we agree with his identification of the database server program as a component, we identify the database along with the database server program as a component instance, not a supported object or a module.

As we have introduced a differentiation between components and component instances in Section 2.3.2, the no-observable-state argument should be reviewed for both of them.

By our definition, a component instance clearly may have its own internal state. We agree that direct exposition of the internal state of a component instance is generally a breach of encapsulation, thus it should be avoided in the design of the component. However, if some states and their transitions are clearly defined in a component's contract, any of its instances should be able to expose these states, which may or may not map to the instance's internal state in a one-to-one manner. Therefore, a component instance may expose its state.

We will regard two component instances C_1 and C_2 of component C as different component instances if they carry different internal states at any point in their lifetime. At any time when C_1 and C_2 have the same internal state, they would appear as copies. However, they still would have to be distinguishable from each other.

As a result of these two arguments, no-observable-state argument does not seem to hold for component instances with respect to our perspective.

In the case of components, an analysis of the no-observable-state argument leads to a component identity discussion. The questions are whether it is possible to have a component with two different states in one system (or process as Szyperski puts it), and whether each state along with a component should be regarded as a different component. Our answer to the first question is af-

firmative, with the definition of component state introduced in Section 2.3.2. With respect to the second question, we prefer to refer to two deployments of one component in a system, since taking them as two different components lead to a discussion of whether they are really different components or not. Regardless of whether they are different components or different deployments, they should be identifiable for the purpose of instantiation. If one chooses to refer to them as different deployments, then such identification can also be considered as exposition of state.

Appendix B

Survey of Network Emulators and Testbeds

As discussed in Section 2.2.3, network emulation is characterized by use of real parts from the system under test (SUT) and its context in the stand-in for the system under test (SIFSUT) in the experimental setup. In an emulation-based experiment in the networking domain, the SUT is the network that is being studied in the experiment.

We should point out that discussions about network emulation that can be found in the literature agrees with our stance, in that it is generally recognized that emulation involves real elements used along with simulators. However, additional properties are usually required. For example, Guruprasad et al. present running in real time also as a defining property of emulators [78]. As discussed at the end of Section 2.2.3, we do not agree that this should be taken as a defining property. Nicol et al. also take running in real time as a defining property [128]. They further classify emulators used for networking research into network emulators and real-time simulators, based on whether time-stepped (a.k.a. time-based) or event-driven approach to time management is used for the simulation in the SIFSUT. Although we agree that time-stepped and event-driven approaches have their own set of advantages and disadvantages when used in an emulation-based experimental setup, which justifies differentiating between the two, their naming is quite confusing: both network emulators and real-time simulators as they define, end up used in emulation-based experimental setups.

In the survey below, which have been published as a part of [73], we will start with the very few systems that we consider as testbeds according to our stance. Then we will look at the emulation systems reported in the literature, starting from systems that have mostly real elements, and then move more or less gradually towards systems that are mostly simulated. This division is not introduced as a proper categorization, but just to make the presentation a bit more tidy: it is easier to discuss what is being simulated for the mostly real emulation systems, and what is being used as real for the mostly simulated ones.

B.1 Network Testbeds

Systems that satisfy the reflexivity property presented in Section 2.2.2, which we have regarded as a defining property for testbeds, are very rare. Ionescu et al. describe NIST*net2, which provides network resources available to the researchers [88]. It is built by connecting dedicated networks in four institutions in Canada. CREATE-NET is a network installation in a rural part of Italy in the autonomous province of Trento [33]. It has both wired and wireless parts. The goal of CREATE-NET is providing a network where researchers can experiment with networked communities. Another testbed is MIT RON (Resilient Overlay Network) [7]. It is a set of hosts distributed over the Internet (36 in 31 different cites as reported in [7]), available to the researchers to use by

acquiring an account. It is similar to PlanetLab, however it is not open to everyone, therefore security is not actively addressed. All hosts run a normal FreeBSD installation.

B.2 Network Emulation – Mostly Real Systems

In certain emulation environments reported in the literature, almost nothing is explicitly simulated. Therefore being considered as a testbed or an emulation environment depends on the definition of the SUT in a given experiment to be conducted using these systems. MIT Roofnet is a good example, where real changing conditions for the wireless channel between the nodes distributed on the roofs of some buildings near the MIT campus is used for testing routing in wireless networks [36]. PlanetLab is very similar to MIT RON in the sense that it is made up of some number of hosts distributed over various sites all over the world. However, the definition of the concept of slides makes the hosts as defined in an experiment somewhat virtual. This fact prevents its categorization as a testbed according to our stance [144].

In the CMU DSR experiment [120], Maltz et al. use the changes of the topology due to mobility of the nodes that are mounted on cars in the campus, as a surrogate for topology changes due to mobility in mobile ad hoc networks (MANETs). In Ad hoc Protocol Evaluation testbed (APE), which is not a testbed according to our definition but an emulation, the style in CMU DSR in simulating topology and mobility is taken one step further and the movements of the people carrying laptops are explicitly choreographed and controlled [115, 130].

There is a good reason why the emulations related to wireless networks mentioned up to this point are trying to use real physical conditions as surrogates for the physical conditions in the SUT and its context: the physical channel is difficult and computationally intensive to simulate accurately. However, it turns out that there is also a need for simulating the physical channel using the physical conditions in the emulation in a more accurate and controlled way. In the Illinois Wireless Wind Tunnel (iWWT), an anechoic chamber is set up, which provides isolation of the experiment environment from the outside RF interference, at the same time providing an anechoic enclosure [181]. The topology is then scaled down by adjusting the transmitting power of the devices that are put into the iWWT. Since the anechoic chamber is designed to be free of unwanted RF, the background noise is added using other transmitters in accordance with the requirements of the experiment. Similar methods for creation of wireless channel effects are also used in ORBIT [137, 108, 67]. However, the positions of the nodes are fixed in ORBIT on a 20x20 matrix, while in iWWT the researchers use small mobile robots.

MiNT (Miniaturized Wireless Network Testbed), which is again not a testbed according to our definition of the term, is another system that uses miniaturization, or scaling down, in order to make use of physical radio communications in emulating MANETs. In MiNT, radio signal attenuators are attached to wireless devices, thereby reducing their transmission range [47]. The nodes are mounted on small mobile robots that are remotely controlled, as in iWWT. An interesting property of the MiNT is that it is reported to work with NS [23], in such a way that the physical layer is simulated using MiNT. However in that case, MiNT becomes a complete simulator, and ceases to have any entities used as real unless the SUT or its context involves mobile robots with signal attenuators attached.

In addition to iWWT and MiNT, other researchers have looked into using attenuators for controlling the wireless channel effects, too. Kaba and Reichle work with unmodified computers and network interface cards (NICs), and attempt to build an environment where the wireless signal propagation is either attenuated, or guided through cables between the communicating NICs [95]. Judd and Steenkiste capture the signals at the antenna, and uses an FPGA based digital signal processor to attenuate signals between the transmitting station and the receiving stations [93, 94].

Their method makes it more possible to repeat wireless physical layer effects across simulation runs, while still using unmodified NICs.

Another focus in building systems to be used for emulation-based experiments is the experiment control mechanisms that would allow multiple researchers to share the resources available. This problem has been targeted in many different systems, such as ORBIT, PlanetLab, and MIT RON. Some of the projects aim only for providing a set of resources for experimenters to build their own emulation-based experimental setup, such as Embedded Wireless Modules (EWM) [156], or the UCLA HNT [174].

B.3 Network Emulation – Mostly Simulated Systems

Some protocols have complex implementations, which makes re-implementing them in a simulator prone to errors and not very accurate. For this reason, various researchers have taken the real protocol implementations in open source operating systems, and packed them in a way that they can be incorporated in simulators. In ENTRAPID, network stack from the FreeBSD is packed in a way that works in the user level, so that the protocol developers can experiment with their own protocols incorporated into the stack without requiring superuser priviledges [86]. However, the processes running on the simulated nodes need slight modification. The topology and the physical layer in ENTRAPID are simulated. Ely et al. have also worked on the same problem, and they have converted the FreeBSD 3.3 protocol stack to work as a library in the user space [55]. Zec and Mikuc modify the protocol stack of 4.4BSD operating system in order to allow multiple independent instances of the stack to exist in the kernel, connected via simulated links [196]. Jansen and McGregor have packed network protocol stacks in Linux, FreeBSD, and OpenBSD as shared libraries, and implemented an NS agent that is capable of using these stacks [91]. They call their approach the Network Simulation Cradle, and say that their approach can be used with other simulators as well. Bless and Doll use OMNeT++ [182] instead of NS, and incorporate the TCP/IP stack from FreeBSD as a simple model[1] in OMNeT++ [19]. They address the problem of synchronizing the kernel timers that are used by the protocol stack with the virtual time in OMNeT++. Furthermore, the function calls to the socket library are represented by messages to be received by the simple model they have developed. Bavier et al. use a different approach, where they implement Click modular routers in slices on PlanetLab hosts, and construct the the stand-in for the network that is the subject of the experiment as an overlay on PlanetLab [14].

In some of the systems that allow use of implementations of protocols as real, it is more difficult to decide whether the experimental setup built using these systems are emulation-based, or pure simulation-based. The reason is that these systems use unusual protocol implementations, but at the same time it is pointed out that they can in fact be used in real systems as well. The JEmu system builds on a four layer protocol stack for MANETs [60]. At the lowest layer in JEmu, which corresponds to the radio communications, the frames are forwarded to a physical layer simulator running on a different host. In [97], Karrer et al. incorporate into NS protocols that are implemented as Click protocol graphs used by the Click modular router [103].

Using real hosts whose traffic is routed through virtual networks appears as another identifiable method. The emulation extension of NS is a typical example [57]. NS is monolithic, but since NS emulation extension simulates whole networks, it is possible to partition the SUT and its context into different networks and assign them to a set of simulators running on different hosts, as done in EmuLab [78]. While NS is monolithic, there are also distributed simulators used in emulation-based experimental setups, such as IP-TNE. IP-TNE is built on IP-TN, which uses CCTKit that

[1] A *simple model* is a component type in OMNeT++ which is not a composite.

implements the Critical Channel Traversing (CCT) algorithm for parallel discrete event simulation [161, 162, 100]. In [21], Bradford et al. discuss different methods for reading packets from and writing packets to real networks for network emulators such as IP-TNE. Another system that uses a discrete event based simulator is RINSE [112], which is built on iSSF (formerly known as DaSSF). RINSE uses what Liljenstam et al. call "multiresolution modeling": background traffic is simulated using fluid models that require less resources to simulate, while the traffic of interest —the foreground traffic— is simulated at the packet level. The target application area of RINSE is network attack preparedness exercises, and therefore it includes some models that are not normally found in other network simulators, such as CPU models. In ModelNet, the environment is divided into two sets of hosts called core nodes and edge nodes [180]. The network that is the subject of the experiment is modeled as a set of pipes, which are assigned to the core nodes. The core nodes then cooperate to subject the traffic to the bandwidth, congestion constraints, latency, and loss profile of the target network topology. The edge nodes are the real hosts whose traffic is routed through the virtual network. While ModelNet is targeted for wired IP networks, MobiNet is an extension of the same approach but it targets MANETs [118]. In addition, MobiNet allows for multiplexing of virtual nodes on the edge nodes.

Another popular approach is the use of traffic shapers, which are placed between the protocol stack and the network device driver in a kernel. This way, the protocol stack and the programs running on top of it are used as real, while the rest of the network is simulated. For example, in NETShaper, flow parameters such as bandwidth and delay are controlled by a user-space program [82, 81]. A similar approach is followed in EMPOWER [200], which targets wired IP networks, and EMWIN [201], which is based on EMPOWER but it targets MANETs. At the extreme case of traffic shaping, it is possible to simulate the presence of the connection between the nodes in a network with only the existence and non-existence of links. As an example, MNE (Mobile Network Emulator) is a distributed system which abstracts away physical layer effects and mobility behind topological changes simulated by IPTABLES based packet filtering controlled from a central controller [117].

An alternative to placing traffic shaper modules in the kernel is the use of the universal TUN/-TAP driver, which is designed for implementing tunneling using user level programs. The emulators NCTUns [185], EmuNet [99], and DINEMO [74] use the TUN virtual network interfaces, which intercept packets after the IP protocol implementation. NEMAN [149] is a similar system, but it uses the TAP virtual network interfaces, which intercepts frames before they are handled to the network driver to be sent to the network. Of these systems, NCTUns and NEMAN are monolithic, while EmuNet and DINEMO are capable of being distributed. In all of these systems, the protocol layers above and including IP, along with the programs communicating over the network, are used as real. Considered from software engineering perspective, DINEMO has the added advantage of being supported by a component model and multiple component platform implementations.

With the developments that allow multiple operating systems to run on one base operating system, another approach has recently become possible. User Mode Linux (UML) provides a Linux that runs in the user mode. UML has been used for implementing virtual nodes that are then connected by a network simulated below the network driver layer. vBET [92], which targets wired networks, and the system developed by Guffens and Bastin [77], which targets MANETs, are examples to using UML. Using micro-kernel based approaches has also been explored, as exemplified by the work by Engel et al. [56], which targets wireless networks.

While it may not be feasible for MANETs or wired networks, simulation of all hardware including the CPU so that unmodified programs can be run, appears to be a feasible technique for emulation-based experiments for sensor networks. ATEMU is one such system that allows different hardware configurations [147]. MEADOWS VMN (Virtual Mote Network) allows multiple

virtual motes per real host that is participating in the emulation [116]. The virtual motes can tun TinyOS, and TinyDB or other applications on top, while hardware of the mote, and sensor and wireless channels are simulated in MEADOWS VMN.

Another original direction is explored by Haeberlen et al. in their system called Monarch [79]. In Monarch, the idea is to use the latency observed at the moment between the host on which the virtual sender and the virtual receiver resides, and a remote host on the Internet. For this purpose, for every packet the virtual sender wants to send, Monarch captures it and sends a probe packet of the same size to a remote host associated with the virtual receiver. When a reply is received, the virtual receiver is allowed to receive the packet. In the direction from virtual receiver to virtual sender, Monarch passes the packets without delay. This way, both the sender and the receiver observes the round-trip-time obtained from the probe packet. Their approach targets transport layer studies only, and can be used with unmodified implementations of transport layer protocols in the Linux kernel.

B.4 More Emulators

The survey presented here is the most encompassing one reported in the literature in terms of number of systems covered, to the best of the author's knowledge at the time of writing. However, no claim is made that it covers every emulation system or testbed. Examples of systems that can be added to this survey include Netbed [189], ONE [6], MobiEmu [199], NIST Net [34], MASSIVE [121], hitbox [3], Delayline [87], SensorSim [140], W-NINE [141], Dummynet [157], ENDE [195], REAL, NEST, PacketStorm, UMLSim, RAMON, TOSSIM, EMStar, and possibly some others.

B.5 Other Surveys

This survey focuses mainly on the techniques used for building emulators for various kinds of networks. Other surveys exist in the literature, such as that by Kropff et al. [104], which focuses on MANETs. A small-scale comparative survey also appears in [47].

Appendix C

Survey of Available Component Models

Brad J. Cox makes the following observation in the preface to the second edition of his book "Object-Oriented Programming: An Evolutionary Approach" [42]:

> And everyone is asking "What could such different technologies possibly have in common? Do they have anything in common? What does "object-oriented" really mean?"

Today, the same quote seem to apply to tools, environments, and programming languages for developing component-based software. In this section, a few of the major component-based approaches are presented, with special focus on their component models.

C.1 CORBA Component Model (CCM)

OMG is a non-profit open consortium with more than 450 members (as of January 2007), whose aim is to provide standards that allow interoperable open systems. The Common Object Request Broker Architecture (CORBA) is the OMG's object-based systems architecture [134]. Since CORBA 3.0, the OMG suite of standards also include a component model built on CORBA, which is called the CORBA Component Model (CCM).

Since CCM is closely related with CORBA, a very brief overview of CORBA will be presented before we start describing CCM. The overall architecture of CORBA is shown in Figure C.1. A client is defined as that which has access to an object's reference, and which invokes operations on that object. Dynamic Invocation Interface (DII) and the Interface Definition Language (IDL) stubs provide the points of access to services provided by various objects defined on the Object Request Broker (ORB). While the IDL stubs are created by the ORB using the IDL descriptions of object interfaces, the DII is used by clients for dynamic construction of object invocations. The ORB core provides the basic representation of objects, and manages and marshalls/unmarshalls the communication requests. The ORB interface includes a few services that is common to all ORB implementations. The majority of the services are provided through the object adapters. It is expected that different object adapter implementations may target a wide range of object granularities, lifetimes, policies, and implementation styles. The IDL skeletons and the Dynamic Skeleton Interface (DSI) allows the object adapters to locate the code that implement particular functionality of defined objects. Like DII and IDL stubs, the IDL skeletons are compiled from static IDL descriptions of interfaces of objects, and DSI is used in order to create skeletons that may make use of dynamic knowledge, such as parameter type information.

The CORBA 3.0 defines an object adapter which is called the Portable Object Adapter (POA). POA is expected to be supported by most CORBA products, with the exception of those that target

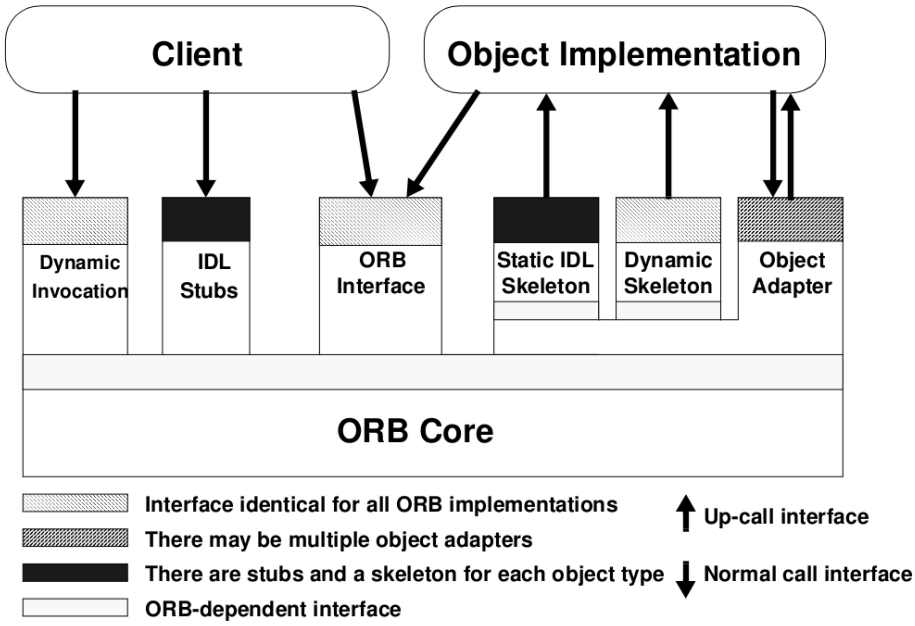

Figure C.1: Abstract architecture of CORBA. Appears as Figure 2-2 in [134].

applications that would require radically different services. POA have replaced its predecessor, the Basic Object Adapter (BOA). The reason why BOA is replaced is because it did not standardize some services that are later found to be useful, which has lead to vendor specific extensions being developed that undermined interoperability.

POA details the CORBA object model by introducing new abstractions such as server, POA objects, managers, and servants. The abstract architecture of POA is shown in Figure C.2.

A server is defined to be a computational context in which the implementation of an object exists. A client is similarly defined to be the computational context from which an object is invoked. Object references are created and exported by servers, to be used by clients.

Servers are composed of POAs, and some special objects. Every POA has an associated POA manager object, and hosts some number of servants. These servants implement a set of services for objects. The mapping between objects and servants is quite flexible: a servant may implement a particular service for a specific set of objects, or for all objects of some type. Furthermore, it is not necessary for all servants that implement an object to reside in the same POA, or server. Servants may be created statically by some initialization code, or dynamically on demand by defining a servant manager object associated with a POA. Similarly, necessary POAs can be created dynamically on demand by defining an adapter activator object. Policies, such as those related to transactions or security, are set using policy objects associated with POAs.

While multi-threading is optional, CORBA specification defines the thread models for the POA in case multi-threading is used. There are three thread models that can be set through a policy object associated with a POA:

Single Thread Model: All POAs are individually single threaded. Note that different servants in different POAs may be implementing particular methods of an object, therefore an object

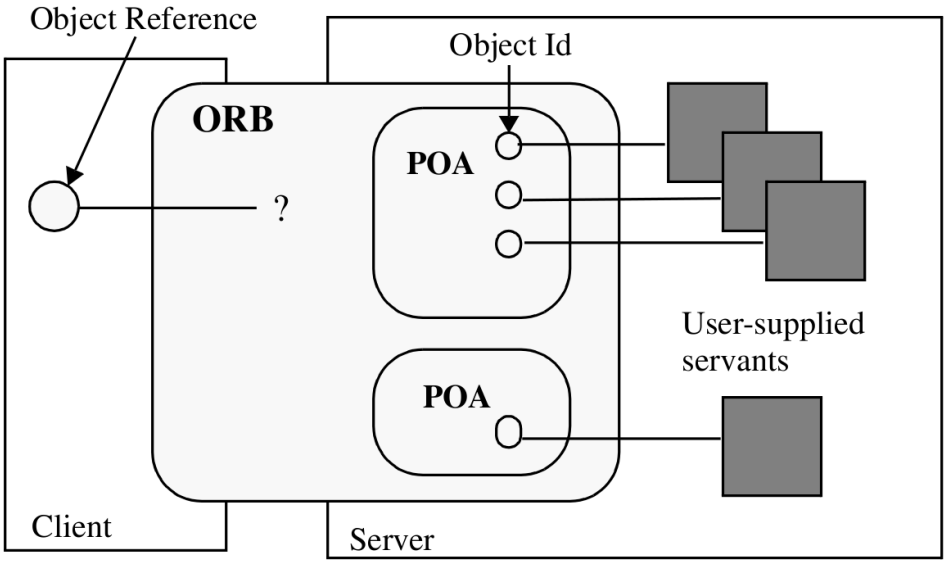

Figure C.2: Abstract architecture of POA. Appears as Figure 11-1 in [134].

might still be distributed over multiple threads. Distinct POAs might be running concurrently.

ORB Controlled Model: ORB/POA controls the use of threads, not the developer of the objects.

Main Thread Model: A set of POAs, designated as the "main-thread POAs" activate their servants in a sequential manner, regardless of whether these POAs are in the same thread/process/host or not.

In any of the three thread models above, further use and control of threads used in implementing the servants in a single POA is left to the developers.

Starting with CORBA 3.0, OMG included a component model called the CORBA Component Model (CCM) into its range of specifications [135]. CCM extends upon CORBA IDL and the interface repository metamodel, and the corresponding implementation framework is designed to be compatible with POA.

There are two levels of components in CCM: basic and extended. The difference between the two levels is mainly based on the different kinds of interaction points, which are called ports, that can be defined for them. While basic components are limited to attributes, extended components may use so called facets, recepticles, event sources and sinks, as well as attributes. The motivation for introducing the concept of basic components is to ease integration with Java Enterprise Java Beans.

There are five different kinds of ports in CCM:

Facets are distinct and named interfaces provided by the component for the clients.

Receptacles are named points of connection for accepting references to facets.

Event sources are categorized into two: publishers and emitters. In the case of publishers, multiple subscribers can receive events created by a publisher, and a channel is dedicated to one publisher component. In contrast, emitters are connected to the consumers of their events in a one-to-one manner, while in the CCM specification, it is defined as possible for multiple emitters to share a channel.

Event sinks represent the consumers of events produced by event sources. They cannot distinguish or choose between available event sources, including whether they are publishers or emitters.

Attributes are named values exposed through accessor and mutator operations.

Component instances of a component type are hosted in one or more component homes, which also provide the equivalent of static members for the component type. Primary keys, which serve as unique ids for component instances, are defined with respect to component homes.

An interesting point, which is related to the design of the MICA component model that is presented in Chapter 4, is that CCM recognizes the need for an explicit configuration phase, and includes optional support for it. The component life cycle can be divided into two mutually exclusive phases, called the configuration and the operation phases, through use of the `configuration_complete` operation defined in CCM for components. However, CCM specification notes that in practice, configuring distributed object assembly proves very difficult. The primary reason appears to be the subtle ordering dependencies that are difficult to discover and enforce.

Composition is done in a context-based manner in CCM. Components are hosted in containers which provide them with an execution context. The container's context provides the services to the component that the component needs in order to function.

CCM defines two threading models to specify control flow: "serialize" and "multithread". The serialize threading policy denotes that the component is not designed to be thread-safe, therefore the container should prevent simultaneous access to the component by multiple threads. A component that is specified to be multithread is capable of handling multiple simultaneous accesses without the help of its container.

C.2 Component Models in Java Suite

Szyperski identifies five component models in Sun's Java suite [173]:

Applets are lightweight components used for augmenting web pages at the client side. They can be composed by placing them on the same web page, and then using the class AppletContext from one applet instance to find another applet instance by name.

Servlets are lightweight components used for augmenting web pages at the server side. They can be implemented using the Java language, or as Java Server Pages (JSP).

JavaBeans specification focuses on components that are composed by wiring. JavaBeans beans are targeted mainly at the client side.

Enterprise Java Beans (EJB) specification focuses on components that are supported and composed through container-integrated services using declared attributes and deployment descriptors.

Application client components are unrestricted applications at the client side, which have access to a Java Naming and Directory Interface service that provides enterprise naming context.

Of these component models, JavaBeans and the Enterprise JavaBeans (EJB) are the most relevant models to this thesis.

JavaBeans beans are component types. They are customized and connected, saved and instantiated to create component instances, which will be referred to as bean instances. JavaBeans beans can distinguish whether they are being used at run time or design time, and can adjust their appearance and functionality dynamically. JavaBeans bean instances, which are customized and connected at design or deployment time, are saved for reloading at the time of application execution.

JavaBeans support a two different ways of communication: beans can communicate by producing and consuming events, which provide an implicit invocation scheme, or by modifying properties of the receiver bean through directly invoking its methods. Events can be communicated in a unicast, n-cast, or multicasting manner. Events are objects, which are preferably immutable. The event passing mechanism between JavaBeans bean instances makes use of the listener pattern, where a set of event consumers register instances of an event listener class with an event producer.

JavaBeans bean properties can be of arbitrary types. These properties are accessed through getter and setter methods, whose names follow a certain pattern and therefore they can easily be discovered using Java's introspection. In addition, changes in the value of a property can be configured to create "change of property" events.

Since events are the main method of communication between JavaBeans bean instances, composition of bean instances is done by wiring them together using a listener pattern. Therefore the composition in JavaBeans is done in composition by wiring style (see Section 2.3.7).

From a control flow perspective, concurrency is considered orthogonal to JavaBeans. Thread execution may follow an event to the listener as a result of calling the receiver's event reception method. This creates a source of potential deadlock, if there are locks that are shared by the event producer bean instance and one or more consumer bean instances.

Containment and services protocol generalizes the JavaBeans model. This protocol allows logically nesting of JavaBeans bean instances, and the container bean instance can assume various services in the Java API for controlling or extending these services. This protocol appears to be a limited support for a kind of EJB style containers.

JavaBeans are packed and distributed in Java Archive (JAR) files. These ZIP archives include the class files, and in addition they may include various resources, and serialized form of some objects to be used as the prototype JavaBeans bean instance for creating new instances by copying it at load or run time. A JAR file may pack more than one JavaBeans beans.

Although they share "JavaBeans" in their names, Enterprise JavaBeans (EJB) provide a different model than JavaBeans, especially in terms of composition. EJB bean instances and homes are placed in containers that provide a context. The context provides services to the EJB bean instances, and it is only the context the EJB bean is supposed to obtain services from. A home roughly corresponds to class level, static methods for an EJB bean (type). EJB beans are packed with what is called a deployment descriptor in their JAR files. The deployment descriptors describe the needs and requirements of the EJB bean instance from its context at run time. Composition of the EJB bean instances is done at deployment by creating a context that satisfies the EJB bean's deployment descriptor. Therefore the EJB beans are composed in context-based composition style (see Section 2.3.7).

The EJB containers serialize all invocations to the EJB bean instance it is encapsulating. Thread creation by the EJB bean instance is not allowed. Containers also protect bean instances from reentrant calls, with the exception of entity type EJB beans that explicitly declare that they can handle re-entrancy. Faults in EJB bean instances are also isolated by their containers, by invalidating and destroying the encapsulated EJB bean instance if it throws an uncaught exception.

There are four different types of EJB beans: stateless session, stateful session, entity, and message-driven beans. Message-driven beans probably are the ones that come most closer to MICA components. They are associated with a single message queue, which receive from multiple message producers. When the container needs to activate the bean instance to process an incoming message, it calls the bean instance's `onMessage` method. Message-driven bean instances are stateless: no state is kept across messages being processed.

Distributed computing support in Java is provided through either Remote Method Invocation (RMI), RMI over CORBA IIOP protocol, or CORBA, and relies heavily on the Java object serialization service. Distributed computing support in EJB containers builds on RMI and RMI over IIOP.

C.3 Microsoft Way: COM and .NET

Component Object Model (COM) is a component model developed by Microsoft [191, 76]. It is the foundation of various technologies available on Microsoft's Windows platforms, such as COM+, Distributed COM (DCOM) [85], Object Linking and Embedding (OLE), and ActiveX. In addition to Microsoft's Windows series operating systems, COM support is also available for Apple Macintosh and various flavors of UNIX.

COM is a binary model. Unlike JavaBeans, it is not bound to a specific language, and unlike CORBA, it does not standardize any bindings to specific languages. Therefore it is referred to as being programming language-independent.

In COM, a component type is called a COM class, and a component instance is called a COM object. A COM class is some code that provides implementation of a set of immutable interfaces, and a COM object is some state with identity. There are various senses in which objects are used in object-oriented programming languages, and the COM way of using these terms does not necessarily coincide with all object-oriented programming languages. For example, COM style is closer to multiple interface inheritance in Java, different from implementation inheritance in C++, while it can be considered in some ways similar to the use of abstract classes in C++, and in fact COM objects can be implemented using multiple objects in these languages. Furthermore, COM objects need not even be coded in an object-oriented programming language at all.

An interface is actually a single pointer to a list of pointers to methods defined in the interface. This table is called a vtable, following its similarity to the virtual method tables in object oriented languages. Clients use COM objects through the references to the interfaces, which are simple pointers that point to the interface. There are no public state variables in a COM object. However, getter and setter methods along with some naming conventions are used in various frameworks built on COM to provide what is generally referred to as "properties." Interfaces are distinguished by Globally Unique Identifiers (GUID) that are associated with their type and version information.

All COM objects have an interface called `IUnknown`, and all interfaces of a COM object support three methods: `QueryInterface`, `AddRef`, and `Release`. The `QueryInterface` method together with the `IUnknown` provides a client the ability to navigate through a COM object's interfaces. The `AddRef` and `Release` methods allow for per interface reference counting, which is used for collaborative garbage collection along with some rules for when to add and remove references, and how to break cyclic dependencies.

COM provides transparent inter-process and remote interoperability between COM objects. This is achieved by in-process proxy COM objects and stubs. This means that all COM interface references and invocations of these interfaces occur in-process. COM infrastructure transparently handles inter-process communication and network related issues.

COM uses class factories along with a library called Component Object Library (COL) in order

to create new COM objects and to address the bootstrap problem. The class factories are themselves normal COM classes, but they produce instances of some COM class. Every COM class has an associated class identifier (CLSID), which is passed to the COL when requesting creation of a COM instance from that class. If COL is used to instantiate a class, it looks up the class in the registration database, which is called the system registry. Depending on whether the server for the COM class being instantiated is registered as in-process or out-of-process, and whether it is on the same machine or on a remote one, the server is launched by either loading a dynamic link library (DLL) when the server is in-process, or by loading an executable on the local or a remote machine and creating proxy and stub COM objects.

COM defines two methods for reuse: containment and aggregation. In containment, only the container COM object has a reference to the contained COM object, and the container COM object uses the contained COM object as any client would in implementing its own interfaces. The containment relation is only logical: the relationship between the outer and the inner COM objects is created by the inner COM object's reference being encapsulated by the outer COM object. Otherwise, there are no special mechanisms supporting containment.

However, aggregation requires special support by COM, therefore the relationship between the inner and outer COM objects in an aggregation is not only logical. In aggregation, the outer COM object exposes the interfaces of the inner COM object (or objects), therefore allowing clients of the outer COM object to invoke inner COM object's (or objects') interfaces directly. This creates subtleties with the interface navigation mechanism, and creates a cyclic reference which should be handled for reference counting based garbage collection.

Unlike JavaBeans (see Section C.2), control flow and threading is not considered to be orthogonal to the COM objects in COM. COM uses a control flow model based on the "Apartment" abstraction in order to express the relationships between COM objects and threads [123, 164]. In this model, COM objects are placed in apartments. According to whether a COM class is implemented in a way that its COM objects will be thread-safe, and what kind of an apartment the COM object that requested the creation is in, various rules govern in which apartment the new COM object will be created in. There are two different flavors of apartments:

Single-threaded Apartment (STA):

The thread that created the apartment is the only source of control flow for the objects placed in that apartment. This means that any communication originating from an object in another apartment is synchronized through the associated thread. For example, in the Microsoft Windows environment, the message infrastructure of the user interface is used by creating an invisible window for each such thread. The thread then polls this message queue in order to get messages and dispatch them to the COM objects in its STA. A process may contain multiple STAs.

Multi-threaded Apartment (MTA):

A multi-threaded apartment contains COM objects for which control flow is provided through one or more threads. Unlike threads associated with MTAs, calls to the COM objects contained in the MTA are not synchronized by threads. Therefore, while the MTA model is simpler in appearance compared to the STA model, the developers need to be careful to implement necessary synchronization methods themselves. This can lead to more efficient use of resources, but at the same time it can be a source of subtle defects. There can be at most one MTA in a process.

A thread associated with an apartment, whether it is an STA or an MTA, cannot be associated with another one.

COM objects normally communicate by procedure calls. Therefore any message-based abstraction has to be implemented on this procedure call structure. While COM was originally supporting only synchronous procedure calls, asynchronous call support was added to COM using asynchronous RPC services introduced with Windows 2000 [158]. This mechanism simplifies implementation of some mechanisms that traditionally require multiple threads.

As mentioned before, there are related technologies that are complementary to or making use of COM, mainly on the Windows platforms. DCOM extends the COM model with interoperation between remote objects. To support discovery and activation of remote COM objects, it introduces a distributed registry of objects, local part of which is called the service control manager (SCM).

Object Linking and Embedding (OLE) is a framework built on COM that provides a component-based document structure. OLE documents can be more than just "documents": they may be applications such as web-based applications. ActiveX is based on OLE and provides lightweight, client-side component-based document architecture for the web.

While COM has been the foundation for various technologies originated from Microsoft, their relatively new technology .NET takes a different form. .NET is a collection of libraries and Application Programming Interfaces (API) that is built on the Common Language Specification (CLS). CLS somewhat resembles the Java Virtual Machine (JVM), and abstracts away the machine specific details of a host by defining a virtual machine. However, while JVM is tightly coupled with the Java programming language, CLS is designed to be higher-level programming language independent, thus there exists various programming language implementations such as Microsoft's Visual Basic .NET, C#, and Managed C++.

Being component-based is not a goal .NET platform is trying to achieve. .NET allows interoperation with COM, and unless there are complex data types that have to be translated or marshalled, this interoperation is reported to be computationally cheap. In addition to the COM interoperability, .NET includes a System.ComponentModel module, which includes definitions of the interface IComponent and the class Component. These definitions provide the type-system basis for developing component models that use context-based composition. Combined with the packaging and deployment units of .NET called "assemblies", the customizable metadata based deployment and Just in Time (JIT) compilation the platform aims to provide, the synchronous and asynchronous messaging support, transactions support, and the event based communication which has direct support in CLS through delegates and the observer pattern [66], .NET can be used to create component-based systems in a way that provides more options and freedom to the developers. However, it should be noted that more freedom is not always what one asks when developing software: more freedom means more decisions, which leads to a heavier development burden. To give an example, .NET control flow is totally orthogonal to components, and require the developers to design the control flow and threading in their system by using the provided thread classes and supported synchronization primitives. While this allows for a considerably large design space and maximizes the domain to which the .NET platform can potentially be applied, control flow design is a tricky business which can become the source of chameleon-like hard-to catch defects.

C.4 Common Component Architecture (CCA)

There is a feeling in the scientific community that the three most well-known component and object interoperability models, namely Microsoft's COM (Section C.3), OMG's CORBA and CORBA CCM (Section C.1), and Sun's JavaBeans and EJB (Section C.2), put the goal of addressing the complexity of software systems in front of the performance issues. While this approach is correct for most software projects, and especially in the business and enterprise computing domains, performance is the most important concern in the scientific computing domain. With this moti-

vation, Common Component Architecture (CCA) forum was launched in 1998 in order to better understand the requirements for using the component-based methods in scientific computing [17].

CCA is the CCA forum's component architecture that targets the high performance computing domain. It aims to provide support for single program multiple data (SPMD) or multiple program multiple data (MPMD) style architectures, heterogeneity in languages and platforms, local and remote components, easy integration, high-performance with minimum communication and synchronization overheads, and an open and simple specification since the target audience consists of scientists, not computing professionals.

The CCA architecturally resembles CORBA to some degree. A component is a computational entity that communicates through its interfaces using a framework, which provides the run-time infrastructure for the components. A framework provides a unique object to each component through which the component receives services from, and exposes its interfaces through. All interfaces are defined using the Scientific Interface Definition Language (SIDL), which has support for scientific data types.

The interfaces of a component, which are called ports, can be declared as "provides" or "uses" ports, through which a component may provide or receive various services. The framework (which is a component platform in our terminology) connects the provides ports to uses ports, and allows dynamic reconfiguration of the topology of connections between the components in a system during execution.

While the framework provides a dedicated `Services` object to a component, a component is required to provide a `Component` interface. The `Component` interface has only a single method, which is used to inform the component of the `Services` object assigned to it by the framework.

The framework provides different service ports, such as `ConnectionEventService`, `BuilderService`, and `ComponentRepository`. The `BuilderService` allows for formation of composite components. It also allows any component to programmatically control the lifecycle operations on components, such as creating, destroying, and connecting.

CCA does not specify anything about control flow. It appears that control flow derives from procedure calls for components in the same process. For remote components, the framework is expected to provide the communication services through various ports.

C.5 Fractal Component Model

The Fractal Component Model (FCM) [25] is a project of the ObjectWeb consortium. Conformance to the FCM is defined through various levels, and many requirements in the FCM specification are optional.

In FCM, components can be composite, and may provide various levels of services. The lowest level components in the FCM, which are called base components, provide services through invocation of their methods. On one level higher, components may provide introspection, thereby letting other components to discover their interfaces, and therefore the services they provide. On the highest level, a component may provide an interface for other components to inspect or modify its sub-components and their interconnection structure.

In addition, FCM specifies how the component instantiation will be done. For this purpose FCM uses factory components, and in order to solve the bootstrapping problem, it requires a generic factory component to be made available by frameworks.

FCM does not providing specifications about control flow, concurrent/sequential execution, or about threading.

C.6 Mozilla XPCOM

Mozilla XPCOM is the plug-in component model and platform that is also used in the now popular Firefox web browser. XPCOM stands for "Cross Platform Component Object Model" [176], and Gecko is a free software web browser layout engine. Gecko SDK is a collection of tools and libraries that features the XPCOM component framework. XPCOM is a framework and a platform for component software development. It aims to enable run-time assembly of components developed and built independently from each other. It separates implementation of a component from its interface. In addition, XPCOM provides tools and libraries for loading and manipulation of components (component management), services for writing modular cross-platform code, versioning support, file abstraction, object message passing, and memory management. The components are in the form of small, reusable binary libraries, such as DLLs on Microsoft Windows and dynamic shared objects (DSO) on Unix, which can include one or more components. When more than one components are packed in a binary library, it is referred to as a module. XPCOM employs programming by contract approach, using interfaces defined as abstract subclasses of a root interface class. The interfaces are reference counted, and a component is expected to delete itself when the reference counts of its interfaces reach zero, meaning that there are no more clients left using the component. All interfaces, contracts, and components have ids. Component creation is supported through the component's factory interface. Interfaces are defined in XPIDL, a variant of OMG's CORBA Interface Definition Language. Type libraries that provide binary implementations of interfaces for languages other than C++ can be compiled from XPIDL (this is called XPConnect). XPCOM also differentiates components, which are instantiated on use, and services, which are components created in singleton design pattern [66].

C.7 Universal Network Objects (UNO)

Universal Network Objects (UNO) [26] is the architecture that forms the basis for StarOffice, OpenOffice.org, and the Sun ONE Webtop. UNO is also available separately from these products. UNO provides interoperability between components written in different languages, and between components in different processes or hosts. One of the main motivations in creating UNO is the inadequate support for exceptions in other component models. As a model, UNO is similar to a blend of simplified COM and CORBA.

Interfaces in UNO are specified using the UNOIDL, which is an IDL that slightly differs from OMG's IDL and Microsoft's MIDL in various issues on inheritance and exceptions. Interfaces of components may contain methods and attributes. UNO requires all interfaces to have three mandatory methods, with similar semantics to their COM counterparts: queryInterface, acquire, and release. With acquire and release, a collaborative garbage collection is implemented using reference counting. As with COM, there exists special rules to break cyclic references. The queryInterface method is used for querying for a certain interface by its name. UNO supports COM style aggregation, where all interfaces of the components that form the aggregation appears as the interfaces of one "master" or "outer" component.

In addition to interfaces, UNO also supports properties. Three types of properties are defined: simple, bound, or constrained. Simple properties can only be observed or modified, bound properties can notify a set of listeners on value change, and changes in constrained properties are vetoable by a set of listeners.

A service in UNO is defined to be a list of mandatory interfaces and their relations. Such a service definition is to serve as an abstract specification for components. Services does not exclusively describe a component: one component may implement multiple services.

Names of services can be used to request creation of a component that provides that service, without caring for which implemented component is actually being instantiated. One such service that exits in every UNO run-time environment is the "servicemanager". The component that realizes component instantiation is the one that implements the "servicemanager" service, and it is created by UNO run-time at startup. In addition to service names, it is also possible to use the name of a component to create an instance of a particular component.

Every service is required to implement interfaces XServiceInfo and XTypeProvider. The name of the component, and the supported service names can be obtained from the XServiceInfo interface. The XTypeProvider interface resembles the IUnknown interface in COM, and provides the list of all interfaces of the component that implements a service.

Interoperation of components in different processes or hosts is supported through the use of a construct called "bridge". Resolving interface references is done using a special method in an interface provided by the environment. Through the resolved interface, a component may invoke methods implemented by a remote component. The bridge implements a proxy that intercepts the method invocation, and forwards it using a UNO specific protocol.

Control flow is defined through four different types of method calls [16]:

Direct call: Blocks the caller, invokes the method using the same thread the method was called from, then returns to caller and the caller continues.

Spawning call: The method is invoked in a new thread. Caller is not blocked.

Asynchronous call: Same as a spawning call, with the addition that all asynchronous calls by a caller is executed sequentially, and in the order they are called. For a given thread, there is at most one other thread that is executing an asynchronous call originating from it at any time.

Synchronous call: A synchronous call is handled like a direct call, but it does not return control to caller until all its asynchronous calls have returned.

When the component whose methods are being invoked is remote, the control flow options available are much more limited. Only synchronous and one-way communications are supported, in which the caller is blocked until callee returns, or the caller does not block, respectively. In fact, one-way communications is disabled by default, and does not seem to be preferred (see Section 3.3.1 in [159]).

C.8 Some Other Component Models

Here we look at some other component models that provide different and interesting constructs. As should be noted, the list presented in this appendix is not meant to be exhaustive of all component models that can be found in the literature.

Dennis et al. describe a component architecture (model) that is called GridCCM [48], which is built on OMG's CCM, and is targeted for Grid computing. In this model, they define what is called parallel components, which host multiple SPMD components. The calls to a parallel component are intercepted at the client side, by adding a software layer between the ORB and the client through compilation of the client with a specially provided compiler. These calls are replaced by multiple calls to the components that the parallel component hosts.

Briot and Meurisse describe their component model called MALEVA agent component model, which is targeted for implementing behaviors of agents in agent-based simulations [24]. MALEVA components communicate using ports. The components are composed by wiring their ports

together, however they do not discuss how wiring is established in their paper. The ports are categorized into control and data ports in order to separate control flow from data flow. Every component have exactly two control ports: in and out. When a component receives a control signal from its control input port, it executes the behavior corresponding to the signal received, and sends out the signal on the control output port.

Appendix D

MICA Services and Callbacks

Cui-Base-Ambassador

Create CUI

SYNOPSIS

Service name: Create CUI

Parameters(in): CUI-CB type (`cui-cb-type`), Id of EU to create CUI in (`in-eu-id`) (optional)

Parameters(out): Id of new CUI created (`id-new-cui`)

Exceptions: CUI-CB type not known, EU-id invalid.

DESCRIPTION

Create a new CUI in the EU with given id `in-eu-id`. The CUI will be composed of the CUI-Base, which will be created from its implementation in the component platform implementation being used, and the CUI customized behavior which will be instantiated using the type information given as parameter `cui-cb-type`.

The parameter `in-eu-id` is optional. If not provided, the new CUI will be created in the EU of the CUI requesting this service.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The component platform must be able to locate the component code implementing the given CUI-CB type. The EU with id `in-eu-id`, if provided, must exist prior to this service request.

POSTCONDITION

A new CUI with customized behavior of given type is created. The CUI is created either in the EU with the given id, or if an EU id wasn't given, in the EU that the CUI which issued this service request is in. The newly created CUI receives the `CUI-Base Created` callback, before this service request returns to caller. Therefore the CUI-id `id-new-cui` returned is usable immediately after this service request.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service reports an exception

- if the EU-id `in-eu-id` is no longer valid, for example if the EU-id have been invalidated by deletion of the EU,
- if the customized behavior code cannot be located by the component platform from the given CUI-CB type.

Cui-Base-Ambassador

Create EM

SYNOPSIS

Service name: Create EM

Parameters(in): EM descriptor (`em-desc`)

Parameters(out): Id of new EM created (`id-new-em`)

Exceptions: EM descriptor problem.

DESCRIPTION

Create a new Execution Manager.

This service is used for creating a new execution manager (EM) on a computational resource described by the given descriptor `em-desc`. The descriptor is component platform dependent. For example, the resource can be a host which will be part of the execution of a distributed component platform, and the descriptor can be the DNS name of the host. Other usage is possible, such as for identifying processors on a multiprocessor computer.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The EM descriptors is the computational resource identifier. It is human-readable, and the reason is that the distribution should be controllable by the users. The given descriptor `em-desc` must refer to a computational resource identifier that can be located by the component platform. Exact specifications of how this is to work is left to component platform implementations.

POSTCONDITION

An EM-id that refers to the EM described by given description `em-desc`, is created. The created EM is initialized so that it will be able to host new EUs. The returned EM-id can be used to create new EUs with `Create EU` service call, as soon as this service request returns to caller.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the given EM descriptor is not usable by the component platform being used.

Cui-Base-Ambassador

Create EU

SYNOPSIS

Service name: Create EU

Parameters(in): Id of EM to create EU in (`in-em-id`) (optional)

Parameters(out): Id of new EU created (`id-new-eu`)

Exceptions: EM-id invalid.

DESCRIPTION

This service is used to create a new Execution Unit (EU) in the given execution manager (EM) with id `in-em-id`. If `in-em-id` is not provided, the new EU is created in the EM that contains the EU that contains the CUI that requested this service.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The EM with id `in-em-id`, if provided, must exist prior to this service request.

POSTCONDITION

A new EU is created. The EU is created either in the EM with the given id, or if an EM id wasn't given, in the EM that the CUI which issued this service request is in. The created EU is initialized so that it will be able to host new UMIs and CUIs. The returned EU-id can be used to create new UMIs and CUIs with `Create CUI` and `Create UMI` service calls, as soon as this service request returns to caller.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the EM-id `in-em-id` is no longer valid, for example if the EM-id have been invalidated by deletion of the EM.

Cui-Base-Ambassador

Create UMI

SYNOPSIS

Service name: Create UMI

Parameters(in): UMI-CB type (`umi-cb-type`), Id of EU to create UMI in (`in-eu-id`) (optional)

Parameters(out): Id of new UMI created (`id-new-umi`)

Exceptions: UMI-CB type not known, EU-id invalid.

DESCRIPTION

Create a new UMI in the EU with given id `in-eu-id`. The UMI will be composed of a UMI-Base, which will be created from its implementation in the component platform implementation being used, and the UMI customized behavior which will be instantiated using the type information given as parameter `umi-cb-type`.

The parameter `in-eu-id` is optional. If not provided, the new UMI will be created in the EU of the UMI requesting this service.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The component platform must be able to locate the component code implementing the given UMI-CB type. The EU with id `in-eu-id`, if provided, must exist prior to this service request.

POSTCONDITION

A new UMI with customized behavior of given type is created. The UMI is created either in the EU with the given id, or if an EU id wasn't given, in the EU that the CUI which issued this service request is in. The newly created UMI receives the `UMI-Base Created` callback, before this service request returns to caller. Therefore the UMI-id `id-new-umi` returned is usable immediately after this service request.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service reports an exception

- if the EU-id `in-eu-id` is no longer valid, for example if the EU-id have been invalidated by deletion of the EU,
- if the customized behavior code cannot be located by the component platform from the given UMI-CB type.

Cui-Base-Ambassador

Delete CUI

SYNOPSIS

Service name: Delete CUI

Parameters(in): Id of CUI to be deleted (`cui-to-delete`).

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Delete the CUI with id `cui-to-delete`. The links that originate from and end in this CUI are also deleted. Any message in transit over these links will be silently dropped. Any messages in transit that are sent using the CUI-id `cui-to-delete` are also silently dropped.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The CUI-id must not be the id of the CUI issuing this service request.

POSTCONDITION

The CUI with given CUI-id `cui-to-delete` is no longer in the run-time, and the destruction routine of its customized behavior, if there is any, has run to completion. Furthermore, all its links have been deleted. All calls that potentially return an exception when given CUI-ids for non-existent CUIs, do return exceptions for the CUI-id `cui-to-delete` immediately after this call returns. The CUI being deleted may receive and process further messages before this service request returns to caller, since the component platform may need time to ensure safe deletion of the CUI, or because communication delays may be present.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service does not return an exception. An exception is not returned in case the CUI-id `cui-to-delete` is not valid, since it already indicates that the postcondition of this service is satisfied.

Cui-Base-Ambassador

Delete EM

SYNOPSIS

Service name: Delete EM

Parameters(in): Id of EM to be deleted (`em-to-delete`).

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Delete the execution manager (EM) with id `em-to-delete`. All the execution units (EUs) that the deleted EM contains are also deleted. This also results in deletion of all the component instances in these EUs, and any link that has these deleted instances as one or both of its ends. Any messages in transit over these links will be silently dropped. Any messages in transit that were sent using the CUI-ids of any CUIs contained in the EUs that are being deleted, are also silently dropped.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback.

POSTCONDITION

The EM with given EM-id `em-to-delete` is no longer in the run-time, and any cleanup about it is performed. Furthermore, all the EUs, all the instances contained by these EUs, and all links connecting those instances to any other instance in the run-time have also been deleted. All calls that potentially return an exception when given EM-ids for non-existent EMs, do return exceptions for the EM-id `em-to-delete` immediately after this call returns. The instances in the EUs in the EM being deleted may receive and process further messages before this service request returns to caller, since the component platform may need time to ensure safe deletion of the EM, or because communication delays may be present.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service does not return an exception. An exception is not returned in case the EM-id `em-to-delete` is not valid, since it already indicates that the postcondition of this service is satisfied.

Cui-Base-Ambassador

Delete EU

SYNOPSIS

Service name: Delete EU

Parameters(in): Id of EU to be deleted (`eu-to-delete`).

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Delete the execution unit (EU) with id `eu-to-delete`. All the component instances in the EU being deleted are also deleted, along with any link that has these deleted instances as one or both of its ends. Any messages in transit over these links will be silently dropped. Any messages in transit that were sent using the CUI-ids of any CUIs contained in the EU that is being deleted, are also silently dropped.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback.

POSTCONDITION

The EU with given EU-id `eu-to-delete` is no longer in the run-time, and any cleanup about it is performed. Furthermore, all the instances contained by the EUs, and all links connecting those instances to any other instance in the run-time have also been deleted. All calls that potentially return an exception when given EU-ids for non-existent EUs, do return exceptions for the EU-id `eu-to-delete` immediately after this call returns. The instances in the EU being deleted may receive and process further messages before this service request returns to caller, since the component platform may need time to ensure safe deletion of the EU, or because communication delays may be present.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service does not return an exception. An exception is not returned in case the EU-id `eu-to-delete` is not valid, since it already indicates that the postcondition of this service is satisfied.

Cui-Base-Ambassador

Delete UMI

SYNOPSIS

Service name: Delete UMI

Parameters(in): Id of UMI to be deleted (`umi-to-delete`).

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Delete the UMI with id `umi-to-delete`. The links that originate from and end in this UMI are also deleted. Any message in transit over these links will be silently dropped.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback.

POSTCONDITION

The UMI with given UMI-id `cui-to-delete` is no longer in the run-time, and the destruction routine of its customized behavior, if there is any, has run to completion. Furthermore, all its links have been deleted. All calls that potentially return an exception when given UMI-ids for non-existent UMIs, do return exceptions for the UMI-id `umi-to-delete` immediately after this call returns. The UMI being deleted may receive and process further messages before this service request returns to caller, since the component platform may need time to ensure safe deletion of the UMI, or because communication delays may be present.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service does not return an exception. An exception is not returned in case the EM-id `em-to-delete` is not valid, since it already indicates that the postcondition of this service is satisfied.

Cui-Base-Ambassador

Get EM List

SYNOPSIS

Service name: Get EM List

Parameters(in): None.

Parameters(out): List of descriptors and ids of EMs present in the run-time.

Exceptions: None.

DESCRIPTION

Returns the list of EM-ids of the EMs that are present at the run-time. The computational resource descriptors can be obtained by querying these EM-ids.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback.

POSTCONDITION

A list of descriptions and EM-ids of all the EMs present in the run-time at the time of this service request is returned.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service does not report any exceptions.

Cui-Base-Ambassador

Get My Id

SYNOPSIS

Service name: Get My Id

Parameters(in): None.

Parameters(out): Id of the CUI issuing this service request `my-id`.

Exceptions: None.

DESCRIPTION

Return the id of the CUI that requested this service.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback.

POSTCONDITION

The instance id of the CUI making this service request is returned in the out-parameter `my-id`.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service does not report any exceptions.

Cui-Base-Ambassador

Link

SYNOPSIS

Service name: Link

Parameters(in): Id of instance the link originates from (`from-id`), the link id to describe the link at the originating instance (`outlink-id`), id of the instance the link ends in (`to-id`), the link id to describe the link at the ending instance (`inlink-id`), type of messages to be transferred on the link (`message-type`)

Parameters(out): None.

Exceptions: `from-id` or `to-id` is not valid, link already is in use, `outlink` or `inlink` is not valid, message type `message-type` cannot be located.

DESCRIPTION

This method creates a link between the two given component instances, with given inlink and outlink ids.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The instances with ids `from-id` and `to-id` must exist prior to this service request. The link ids `outlink-id` and `inlink-id` should not have previously been associated with another link. The component platform must be able to locate the necessary definitions or implementation for the given message type (`message-type`).

POSTCONDITION

The link described by the 5-tuple (`from-id`, `outlink`, `to-id`, `inlink`, `message-type`) is set up. Immediately after this service returns to the caller, the instance with id `from-id` can use the service requests `Send Message to Link` or `Send Message` depending on whether it is a CUI or a UMI. Furthermore, any further service requests for `Link` that involves (`from-id`, `outlink`) or (`to-id`, `inlink`) pairs would fail with respective exceptions.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception:

- if any of the instance ids `from-id` or `to-id` are no longer valid, for example if any of these instance ids have been invalidated by deletion of the instances they refer to,
- if the pair (`from-id`, `outlink`) or (`to-id`, `inlink`) have previously been used in a `Link` service request,

- if any of the link ids `outlink` or `inlink` is not a valid link id,
- if the component platform is not able to locate the definition or implementation of the given message type `message-type`.

Cui-Base-Ambassador

Register Message Type

SYNOPSIS

Service name: Register Message Type

Parameters(in): Message type descriptor `msg-type`.

Parameters(out): Message type identifier for given message type descriptor `msg-type-id`

Exceptions: Message type described by `msg-type` is not known.

DESCRIPTION

Register a message type that will be used by the CUI requesting this service, and return to it a message type id object `msg-type-id`. This method must be called for each message type that the CUI will be using. Such registrations might be used by some component platform implementations in order to load the implementations of the message types.

For the messages received from a link, this message type id object can be used for checking the type of the message. Such checks might be omitted by some component platform implementations for performance reasons, especially for the case of links that do not cross EM or EU boundaries.

For the messages received that were sent using a CUI-id, this message type id is more useful since the response behavior would presumably depend on the type of the message received.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The component platform should be able to locate the implementation of the message type described by `msg-type`.

POSTCONDITION

The message type information is located and made ready by the component platform, for the instance making this service request to be able create and use messages of this type. A message type identifier `msg-type-id` is returned to be used by the instance for subsequent use, such as testing types of messages received.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the component platform is unable to locate information about the message type description of which is given via parameter `msg-type`.

Cui-Base-Ambassador

Replace UMI-CB

SYNOPSIS

Service name: Replace UMI-CB

Parameters(in): Type of new UMI-CB (`umi-cb-type`), Id of UMI (`umi-id`) to replace the CB of

Parameters(out): None.

Exceptions: UMI-CB type not known, UMI-id invalid.

DESCRIPTION

Replace the customized behavior object of the UMI with given id `umi-id`, without changing its UMI-id. This method allows mechanisms to be developed for simulator interoperability, and just-before-run model replacements.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The component platform must be able to locate the component code implementing the given UMI-CB type `umi-cb-type`. The `Create UMI` service request that created the given UMI with id `umi-id`, which was not necessarily invoked by the CUI instance invoking this service request, must have successfully completed its execution prior to this service request.

POSTCONDITION

The customized behavior of the UMI with given id `umi-id` is deleted. A new CB of given type `umi-cb-type` is created and associated with the base of the UMI with given id `umi-id`. The newly created UMI-CB receives `UMI-Base Created` callback before this service request reports completion. The links previously set up for the UMI with given id `umi-id` are kept intact.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service reports an exception

- if the UMI-id `umi-id` is no longer valid, for example if the UMI-id have been invalidated by deletion of the UMI,
- if the customized behavior code cannot be located by the component platform from the given UMI-CB type.

Cui-Base-Ambassador

Send Message to CUI

SYNOPSIS

Service name: Send Message to CUI

Parameters(in): Id of the CUI to send the message to (`target-id`), the message `msg` to send.

Parameters(out): None.

Exceptions: CUI with id `target-id` does not exist.

DESCRIPTION

Send a message to another CUI, without having a link set up. This method is provided in order to prevent temporary or infrequently used links being set up between CUIs in a system. Since CUIs are capable of using CUI-ids, they can use such ids to exchange messages using this method.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The instance with id `target-id` must exist prior to this service request. The message type of message `msg` should have previously been registered by the CUI issuing this service request, by a `Register Message Type` service request.

POSTCONDITION

The message is accepted for delivery to the instance with id `target-id`, and it will arrive eventually, with a `Receive Message from CUI` callback. The message `msg` is going to arrive at the other end of the link before any messages subject to subsequent calls to this service request with the same instance id `target-id`. The instance making this service request has no control over the message `msg` once this call returns. This would mean that freeing the resources allocated to the message, if necessary, is the responsibility of the receiver.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the instance id `target-id` is no longer valid, for example if the instance id has been invalidated by deletion of the instance.

Cui-Base-Ambassador

Send Message to Link

SYNOPSIS

Service name: Send Message to Link

Parameters(in): A link id `outlink` that describes the link to send the message out to, the message `msg` to send

Parameters(out): None.

Exceptions: Link does not exist, message type mismatch.

DESCRIPTION

Send a message to an outlink. The message is received by the instance at the ending side of the link, with the inlink-id that was set when the link was set up.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The link id `outlink` should have been used in a previous successful `Link` service request along with the id of the CUI making this service call as the originating instance, as the link id that describes the link at the originating instance side. The message `msg` should be of message type given in the same `Link` service call just mentioned. The message type of message `msg` should have previously been registered by the CUI issuing this service request, by a `Register Message Type` service request.

POSTCONDITION

The message is accepted for delivery to the other end of the link, and it will arrive eventually, with a `Receive Message` or `Receive Message from Link` callbacks at the instance whose id was used to describe the ending instance in the `Link` service call that set up the link. The message `msg` is going to arrive at the other end of the link before any messages subject to subsequent calls to this service request in this CUI with the same link id `outlink`. The instance making this service request has no control over the message `msg` once this call returns. This would mean that freeing the resources allocated to the message, if necessary, is the responsibility of the receiver.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the link id `outlink` and instance id of the instance issuing this service call haven't previously been used in a successful `Link` service request as the originating instance id (`from-id` parameter in `Link`), and link id that describes the link at the originating side (`outlink` parameter in `Link`).

- if the message type of the message `msg` is not of type registered when the link was created with a `Link` service request.

Cui-Base-Ambassador

Unlink

SYNOPSIS

Service name: Unlink

Parameters(in): The id of the instance the link originates from (`from-id`), the link id to describe the link at the originating instance (`outlink-id`), the id of the instance the link ends in (`to-id`), and the link id to describe the link at the ending instance (`inlink-id`)

Parameters(out): None.

Exceptions: Link does not exist.

DESCRIPTION

This method deletes a link between two instances.

While both the pair <`from-id`, `outlink-id`> and the pair <`to-id`, `inlink-id`> uniquely identify the link, this method requires both pairs to be provided. The reason is to catch race conditions or information inconsistencies among CUIs in a system, where the link is already deleted and replaced by another link. The message type of the link is ignored, since it is considered to be unlikely that a link would be replaced by a new link that differs only in its message type.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `CUI-Base Created` callback. The instances with ids `from-id` and `to-id` must exist prior to this service request. The instance ids `from-id` and `to-id`, and the link ids `outlink-id` and `inlink-id` should have previously been associated with each other through a `Link` service request that succeeded.

POSTCONDITION

The link described by the 4-tuple (`from-id`, `outlink`, `to-id`, `inlink`) is deleted. Immediately after this service returns to the caller, a `Send Message to Link` or `Send Message` service request by the instance with id `from-id` would fail with an exception. Another link might already been set up by other instances (though it probably would be unlikely) that involve the pair (`to-id`, `inlink`), before this service request returns control to its caller.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the link described by the 4-tuple (`from-id`, `outlink`, `to-id`, `inlink`) does not exist, probably because either the instance with id `from-id` or `to-id`, or the link itself is previously deleted.

Umi-Base-Ambassador

Register Message Type

SYNOPSIS

Service name: Register Message Type

Parameters(in): Message type descriptor `msg-type`.

Parameters(out): Message type identifier for given message type descriptor `msg-type-id`

Exceptions: Message type described by `msg-type` is not known.

DESCRIPTION

Register a message type that will be used by the UMI requesting this service, and return to it a message type id object `msg-type-id`. This method must be called for each message type that the UMI will be using. Such registrations might be used by some component platform implementations in order to load the implementations of the message types.

Message type id that is received by the UMI, can be used for checking the type of messages received from the links. Such checks might be omitted by some component platform implementations for performance reasons, especially for the case of links that do not cross EM or EU boundaries.

PRECONDITION

The CUI making this service call must have received, but not necessarily completed, the `UMI-Base Created` callback. The component platform should be able to locate information about the message type described by `msg-type`.

POSTCONDITION

The message type information described by the message type descriptor `msg-type` is located and made ready by the component platform, for the instance making this service request to be able create and use messages of this type. A message type identifier `msg-type-id` is returned to be used by the instance for subsequent use, such as testing types of messages received.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the component platform is unable to locate information about the message type description of which is given via parameter `msg-type`.

Umi-Base-Ambassador

Send Message

SYNOPSIS

Service name: Send Message

Parameters(in): A link id `outlink` that describes the link to send the message out to, the message `msg` to send.

Parameters(out): None.

Exceptions: Link does not exist, message type mismatch.

DESCRIPTION

Send a message to an outlink. The message is received by the instance at the ending side of the link, with the inlink-id that was set when the link was set up.

PRECONDITION

The UMI making this service call must have received, but not necessarily completed, the `UMI-Base Created` callback. The link id `outlink` should have been used by a CUI in a previous successful `Link` service request along with the id of the UMI making this service call as the originating instance, as the link id that describes the link at the originating instance side. The message `msg` should be of message type given in the same `Link` service call just mentioned. The message type of message `msg` should have previously been registered by the UMI issuing this service request, by a `Register Message Type` service request.

POSTCONDITION

The message is accepted for delivery to the other end of the link, and it will arrive eventually, with a `Receive Message` or `Receive Message from Link` callbacks at the instance whose id was used to describe the ending instance in the `Link` service call that set up the link. The message `msg` is going to arrive at the other end of the link before any messages subject to subsequent calls to this service request in this UMI with the same link id `outlink`. The instance making this service request has no control over the message `msg` once this call returns. This would mean that freeing the resources allocated to the message, if necessary, is the responsibility of the receiver.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this service only reports an exception

- if the link id `outlink` and instance id of the instance issuing this service call haven't previously been used by a CUI in a successful `Link` service request as the originating instance id (`from-id` parameter in `Link`), and link id that describes the link at the originating side (`outlink` parameter in `Link`).

- if the message type of the message `msg` is not of type registered when the link was created with a `Link` service request.

Cui-CB-Ambassador

CUI-Base Created

SYNOPSIS

Service name: CUI-Base Created

Parameters(in): CUI-Base ambassador `cui-base-amb`

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Callback indicating that this CUI-CB is now associated with a CUI-Base.

This method is called by the component platform in order to inform the CUI customized behavior (CB) object about the CUI-Base object it is associated with.

This method is a good place to put the message registration calls.

PRECONDITION

The CUI-Base ambassador assigned to the customized behavior part of the instance is ready to accept service requests.

POSTCONDITION

The customized behavior part of the CUI know how to reach the CUI-Base assigned to it, using the CUI-base ambassador `cui-base-amb`.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this callback should not return an exception.

Cui-CB-Ambassador

Receive Message From CUI

SYNOPSIS

Service name: Receive Message From CUI

Parameters(in): The CUI id of the CUI that sent the message (`sender-id`), the message that is received (`msg`)

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Process a received message, which was sent using a CUI-id.

This method is called by the component platform in order to let the CUI customized behavior (CB) object to process a received message, which was sent using a CUI-id (not over a link). This CUI-CB processes the message, and returns from this method in order to give control back to the component platform, for other component instances in the same EU to receive messages they have received (if any).

PRECONDITION

The instance that will receive this callback should have registered the type of the message `msg` via issuing a `Register Message Type` service request. This also implies that it should at least must have completed its `CUI-Base Created` callback.

POSTCONDITION

Control of any resources related to the message `msg` is transferred to the customized behavior part of the instance receiving this callback.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this callback should not return an exception.

Cui-CB-Ambassador

Receive Message From Inlink

SYNOPSIS

Service name: Receive Message From Inlink

Parameters(in): The link id of the incoming link the message was received from (`inlink`), the message that is received (`msg`)

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Process a received message.

This method is called by the component platform in order to let the CUI customized behavior (CB) object to process a received message. This CUI-CB processes the message, and returns from this method in order to give control back to the component platform, for other component instances in the same EU to receive messages they have received (if any).

PRECONDITION

The instance that will receive this callback should have registered the type of the message `msg` via issuing a `Register Message Type` service request. This also implies that it should at least must have completed its `CUI-Base Created` callback. The CUI id of this receiving instance should have been used in a previous call to a `Link` service request call, along with the link id `inlink` as the describing the receiving end of a link.

POSTCONDITION

Control of any resources related to the message `msg` is transferred to the customized behavior part of the instance receiving this callback.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this callback should not return an exception.

Umi-CB-Ambassador

Receive Message

SYNOPSIS

Service name: Receive Message

Parameters(in): The link id of the incoming link the message was received from (`inlink`), the message that is received (`msg`)

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Process a received message.

This method is called by the component platform in order to let the UMI customized behavior (CB) object to process a received message. This UMI-CB processes the message, and returns from this method in order to give control back to the component platform, for other component instances in the same EU to receive messages they have received (if any).

PRECONDITION

The instance that will receive this callback should have registered the type of the message `msg` via issuing a `Register Message Type` service request. This also implies that it should at least must have completed its `UMI-Base Created` callback. The UMI id of this receiving instance should have been used in a previous call to a `Link` service request call, along with the link id `inlink` as the describing the receiving end of a link.

POSTCONDITION

Control of any resources related to the message `msg` is transferred to the customized behavior part of the instance receiving this callback.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this callback should not return an exception.

Umi-CB-Ambassador

UMI-Base Created

SYNOPSIS

Service name: UMI-Base Created

Parameters(in): UMI-Base ambassador `umi-base-amb`

Parameters(out): None.

Exceptions: None.

DESCRIPTION

Callback indicating that this UMI-CB is now associated with a UMI-Base.

This method is called by the component platform in order to inform the UMI customized behavior (CB) object about the UMI-Base object it is associated with.

This method is a good place to put the message registration calls.

PRECONDITION

The UMI-Base ambassador assigned to the customized behavior part of the instance is ready to accept service requests.

POSTCONDITION

The customized behavior part of the UMI know how to reach the UMI-Base assigned to it, using the UMI-base ambassador `umi-base-amb`.

EXCEPTIONS

Aside from any unexpected exceptions to be determined by the component platform implementation, this callback should not return an exception.

Appendix E

C++ Application Programming Interface for MICA

CUI Base Ambassador

```
class CuiBaseAmbassador {
 public:
  virtual Exception*
    mfSendMessageToLink(const LinkId &outlink,
                        Message* msg) throw() = 0;
  virtual Exception*
    mfSendMessageToCui(const CuiId *target,
                       Message* msg) throw() = 0;
  virtual Exception* mfCreateUmi(UmiId** idOfNewUmi,
                                 const char* umicbtype,
                                 const EuId* where = 0)
    throw() = 0;
  virtual Exception* mfCreateCui(CuiId** idOfNewCui,
                                 const char* cuicbtype,
                                 const EuId* where = 0)
    throw() = 0;
  virtual Exception* mfDeleteUmi(UmiId* uid) throw() = 0;
  virtual Exception* mfDeleteCui(CuiId* cid) throw() = 0;
  virtual Exception* mfLink(const InstanceId *from,
                            const LinkId &outlink,
                            const InstanceId *to,
                            const LinkId &inlink,
                            const MessageType* mt)
    throw() = 0;
  virtual Exception* mfUnlink(const InstanceId *from,
                              const LinkId &outlink,
                              const InstanceId *to,
                              const LinkId &inlink)
    throw() = 0;
  virtual Exception* mfGetMyId(const CuiId** idOfThisCui)
    const throw() = 0;
  virtual Exception* mfCreateEm(EmId** idOfCreatedEm,
                                const char* descriptor)
    throw() = 0;
  virtual Exception* mfCreateEu(EuId** idOfCreatedEu,
                                const EmId* emid = 0)
    throw() = 0;
  virtual Exception* mfDeleteEm(EmId* emid) throw() = 0;
  virtual Exception* mfDeleteEu(EuId* euid) throw() = 0;
  virtual Exception* mfGetEmList(unsigned int* numOfEms,
                                 EmId*** emList) const
    throw() = 0;
  virtual Exception*
    mfRegisterMessageType(const MessageType** mt,
                          const char* messageType)
    throw() = 0;
  virtual ~CuiBaseAmbassador() throw();
```

```
}; // end of class CuiBaseAmbassador
```

CUI CB Ambassador

```
class CuiCBAmbassador {
 public:
  virtual void mfCuiBaseCreated(CuiBaseAmbassador* cba)
    throw() = 0;
  virtual void mfReceiveMessage(const LinkId &inlink,
                                Message* msg)
    throw() = 0;
  virtual void mfReceiveMessage(CuiId *from,
                                Message* msg)
    throw() = 0;
  virtual ~CuiCBAmbassador() throw();
}; // end of class CuiCBAmbassador
```

EM Identifier

```
class EmId {
 public:
  virtual ~EmId() throw();
  virtual bool operator ==(const EmId &rhs)
    const throw() = 0;
  virtual Exception* mfClone(EmId** newId)
    const throw() = 0;
  virtual Exception* mfGetEmDescriptor(char** desc)
    const throw() = 0;
  virtual Exception* mfSerialize(char** data,
                                 unsigned int* size)
    const throw() = 0;
  virtual Exception* mfSerialize(char* data)
    const throw() = 0;
  static void msfSerializationSize(unsigned int* size)
    throw();
  static void msfDeserializeId(EmId** id, const char* data)
    throw();
 protected:
 private:
}; // end of class EmId
```

EU Identifier

```
class EuId {
 public:
  virtual ~EuId() throw();
  virtual bool operator ==(const EuId &rhs)
    const throw() = 0;
  virtual Exception* mfClone(EuId** newId)
    const throw() = 0;
  virtual Exception* mfSerialize(char** data,
                                 unsigned int* size)
    const throw() = 0;
  virtual Exception* mfSerialize(char* data)
    const throw() = 0;
  static void msfSerializationSize(unsigned int* size)
    throw();
  static void msfDeserializeId(EuId** id, const char* data)
    throw();
 protected:
 private:
}; // end of class EuId
```

Exception

```
class Exception {
 public:
  enum ExceptionType {
    EXC_LINK_ALREADY_IN_USE,
    EXC_LINK_NOT_FOUND,
    EXC_EM_DESCRIPTION_PROBLEM,
    EXC_SERVICE_REQ_FAIL,
    EXC_ID_NO_LONGER_VALID };

  Exception(const ExceptionType& type) throw();
  Exception(const ExceptionType& type, const char* msg)
    throw();
  Exception(const Exception& e) throw();
  virtual ~Exception() throw();
  virtual void mfGetMsg(char** msg) const throw();
  virtual void mfGetType(ExceptionType* type)
    const throw();
 protected:
 private:
  const ExceptionType mType;
  char* mMsg;
}; // end of class Exception
```

Instance Identifier

```
class CuiId : public InstanceId {
 public:
  virtual ~CuiId() throw();
  virtual bool operator ==(const CuiId &rhs)
    const throw() = 0;
  virtual Exception* mfClone(CuiId** newId)
    const throw() = 0;
  virtual Exception* mfSerialize(char** data,
                                 unsigned int* size)
    const throw() = 0;
  virtual Exception* mfSerialize(char* data)
    const throw() = 0;
  static void msfSerializationSize(unsigned int* size)
    throw();
  static void msfDeserializeId(CuiId** id,
                               const char* data)
    throw();
 protected:
 private:
}; // end of class CuiId
```

Link Identifier

```
class LinkId {
 public:
  LinkId(const unsigned int &id) throw();
  LinkId(const LinkId &lid) throw();

  operator unsigned int () const throw();
  LinkId& operator = (const LinkId &rval) throw();
  bool operator < (const LinkId &rval) const throw();
 protected:
 private:
}; // end of class LinkId
```

Message

```
class Message {
 public:
  virtual ~Message() throw();
  virtual void mfGetMessageType(const MessageType** type)
    throw() = 0;
  virtual void mfSerialize(char** data, unsigned int* size)
    throw() = 0;
  virtual void mfDeserialize(const char* data,
                             const unsigned int size)
    throw() = 0;
 protected:
 private:
}; // end of class Message
```

Message Type Identifier (Object Form)

```
class MessageType {
 public:
  virtual ~MessageType();
  virtual bool operator ==(const MessageType& mt)
    const throw() = 0;
};
```

UMI Base Ambassador

```
class UmiBaseAmbassador {
 public:
  virtual ~UmiBaseAmbassador() throw();
  virtual Exception* mfSendMessage(const LinkId &outlink,
                                   Message* msg)
    throw() = 0;
  virtual Exception*
    mfRegisterMessageType(const MessageType** mt,
                          const char* messageType)
    throw() = 0;
 protected:
 private:
}; // end of class UmiBaseAmbassador
```

UMI CB Ambassador

```
class UmiCBAmbassador {
 public:
  virtual ~UmiCBAmbassador() throw();
  virtual void mfUmiBaseCreated(UmiBaseAmbassador *uba)
    throw() = 0;
  virtual void mfReceiveMessage(const LinkId &inlink,
                                Message* msg)
    throw() = 0;
 protected:
 private:
}; // end of class UmiCBAmbassador
```

UMI Identifier

```
class UmiId : public InstanceId {
 public:
  virtual ~UmiId() throw();
  virtual bool operator ==(const UmiId &rhs)
    const throw() = 0;
  virtual Exception* mfClone(UmiId** newId)
    const throw() = 0;
  virtual Exception* mfSerialize(char** data,
                                 unsigned int* size)
    const throw() = 0;
  virtual Exception* mfSerialize(char* data)
    const throw() = 0;
  static void msfSerializationSize(unsigned int* size)
    throw();
  static void msfDeserializeId(UmiId** id, const char* data)
    throw();
 protected:
 private:
}; // end of class UmiId
```

Appendix F

Contracts of DINEMO Components

TUN UM Component

Description

Instances of TUN UM components create a TUN virtual network interface, and control the data flowing through it. This component uses a Select Hub UM component to wait for events about the file descriptor associated with the TUN interface. On deletion of the CB part, the destructor of this component destroys the TUN interface it created.

Message Types

Start Message (`startmsg`)**:**
Upon reception of a start message, the component instance creates a new TUN interface, and registers the file descriptor for catching read events by sending an `shfdstatemsg` to the Select Hub UMI it is connected to.

IP Packet Message (`ippacketmsg`)**:**
An IP packet message is created in response to a packet received from the TUN interface through the associated file descriptor, and sent out to the component instance connected on outlink 2. If an IP packet message is received from inlink 2, the IP packet in the message is sent out to the TUN interface.

Select Hub State Message (`shfdstatemsg`)**:**
Select hub state messages are used for registering the file descriptor with the Select Hub UMI, for receiving events that signify more data is available on the file descriptor for reading.

Select Hub Event Message (`shfdeventmsg`)**:**
Select hub event messages are sent by the Select Hub UMI, when there is more data to be read through the file descriptor, which is the one associated with the TUN interface in the case of a TUN UM component instance.

Connection Map — Inlinks

0 (`startmsg`)**:**
Inlink 0 is the link the start message is expected to arrive from. Most probably it would be connected to the first CUI of the simulator.

1 (`shfdeventmsg`)**:**
Inlink 1 should be connected to a Select Hub UMI. Inlink 1 and outlink 1 should form a control and event reception pair for a single file descriptor at the Select Hub UMI side.

2 (`ippacketmsg`)**:**
The IP packets to be sent out to the TUN interface are expected to arrive from inlink 2.

Connection Map — Outlinks

1 (`shfdstatemsg`)**:**

Outlink 1 should be connected to a Select Hub UMI. Inlink 1 and outlink 1 should form a control and event reception pair for a single file descriptor at the Select Hub UMI side.

2 (`ippacketmsg`)**:**

The IP packets received from the TUN interface are sent as IP packet messages to whomever is connected at the other end of outlink 2.

TAP UM Component

Description

Instances of TAP UM components create a TAP virtual network interface, and control the data flowing through it. This component uses a Select Hub UM component to wait for events about the file descriptor associated with the TAP interface. On deletion of the CB part, the destructor of this component destroys the TAP interface it created.

Message Types

Start Message (`startmsg`)**:**
Upon reception of a start message, the component instance creates a new TAP interface, and registers the file descriptor for catching read events by sending an `shfdstatemsg` to the Select Hub UMI it is connected to.

Ethernet Frame Message (`ethernetframemsg`)**:**
An Ethernet frame message contains an Ethernet frame. An Ethernet frame message is created in response to a frame received from the TAP interface through the associated file descriptor, and sent out to the component instance connected on outlink 2. If an Ethernet frame message is received from inlink 2, the Ethernet frame in the message is sent out to the TAP interface.

Select Hub State Message (`shfdstatemsg`)**:**
Select hub state messages are used for registering the file descriptor with the Select Hub UMI, for receiving events that signify more data is available on the file descriptor for reading.

Select Hub Event Message (`shfdeventmsg`)**:**
Select hub event messages are sent by the Select Hub UMI, when there is more data to be read through the file descriptor, which is the one associated with the TAP interface in the case of a TAP UM component instance.

Connection Map — Inlinks

0 (`startmsg`)**:**
Inlink 0 is the link the start message is expected to arrive from. Most probably it would be connected to the first CUI of the simulator.

1 (`shfdeventmsg`)**:**
Inlink 1 should be connected to a Select Hub UMI. Inlink 1 and outlink 1 should form a control and event reception pair for a single file descriptor at the Select Hub UMI side.

2 (`ethernetframemsg`)**:**
The Ethernet frames to be sent out to the TAP interface are expected to arrive from inlink 2 in Ethernet frame messages.

Connection Map — Outlinks

1 (`shfdstatemsg`)**:**

Outlink 1 should be connected to a Select Hub UMI. Inlink 1 and outlink 1 should form a control and event reception pair for a single file descriptor at the Select Hub UMI side.

2 (`ethernetframemsg`)**:**

The Ethernet frames received from the TAP interface are sent as IP packet messages to whomever is connected at the other end of outlink 2.

Router UM Component

Description

Instances of Router UM components simulate the routing functionality in the network layer. In the direction of higher level protocols to lower layers, a Router UMI finds the IP address information of the next hop and puts it together with the IP packet. In the other direction, the Router passes the IP packet to the upper layers if the IP address is the address of the node, or sends it downwards to be sent to its next hop.

Message Types

IP Packet Message (`ippacketmsg`)**:**
IP packet messages contain the data related to a single IP packet.

IP Packet With Next Hop Message (`ippktwnexthopmsg`)**:**
These messages contain an IP packet, along with the IP address of the next hop the IP packet should be sent to.

Route Configuration Message (`routeconfigmsg`)**:**
These messages contain a target IP address, along with the IP address of the next hop for packets to be sent to that target IP address. A route configuration message is used for modifying the routing table, either for adding a route or deleting it.

Connection Map — Inlinks

0:
Inlink 0 is reserved for configuration messages for the Router UMI.

1 (`ippacketmsg`)**:**
Inlink 1 is to be connected to the upper layer protocols in the simulated node, typically a TUN UMI. When an IP packet is received from inlink 1, the Router UMI decides the IP address of the next hop the packet should be sent, and sends the IP packet and next hop IP address pair to outlink 1.

2 (`ippacketmsg`)**:**
Inlink 2 is to be connected to the lower layer protocols in the simulated node, typically an ARP UMI. When an IP packet is received from inlink 2, the Router UMI first checks whether the destination IP of the packet is the IP address of the simulated node. If the packet is destined for the simulated node, the IP packet message is sent out on outlink 2 for the upper layer protocols. If the destination IP is different, it is forwarded to its next hop by producing an `ippktwnexthopmsg` message on outlink 1.

3 (`routeconfigmsg`)**:**
Messages that request addition or deletion of routes are expected on inlink 3.

Connection Map — Outlinks

1 (`ippktwnexthopmsg`)**:**

Outlink 1 is to be connected to the lower layer protocols in the simulated node, typically an ARP UMI. The IP packets to be sent out to the simulated network, and their next hop IP addresses are output on this outlink.

2 (`ippacketmsg`)**:**

Outlink 2 is to be connected to the upper layer protocols in the simulated node, typically a TUN UMI. The IP packets that are destined to the simulated node are output on this outlink.

ARP UM Component

Description

Instances of ARP UM components convert the next hop IP address of an IP packet being sent to the network by their simulated node, generate an Ethernet frame destined to the MAC address associated with the next hop IP address, and sends the Ethernet frame to the lower layers. They also generate ARP queries, and reply incoming queries from the simulated network.

Message Types

IP Packet Message (`ippacketmsg`)**:**
IP packet messages contain the data related to a single IP packet.

IP Packet With Next Hop Message (`ippktwnexthopmsg`)**:**
These messages contain an IP packet, along with the IP address of the next hop the IP packet should be sent to.

Ethernet Frame Message (`ethernetframemsg`)**:**
An Ethernet frame message contains an Ethernet frame.

Connection Map — Inlinks

0:
Inlink 0 is reserved for configuration messages for the ARP UMI.

1 (`ippktwnexthopmsg`)**:**
Inlink 1 is to be connected to the upper layer protocols in the simulated node, typically a Router UMI. When an IP packet along with the IP address of its next hop is received in an `ippktwnexthopmsg` message, the ARP UMI attempts to find the MAC address of the next hop IP address. If the address is found, the IP packet is put in an Ethernet frame as payload, and the Ethernet frame is sent to the simulated network with the next hop's MAC address as its destination. If the MAC address of the next hop is not known to the ARP UMI, it drops the packet and takes necessary steps for finding the MAC address of the next hop from its IP.

2 (`ethernetframemsg`)**:**
Inlink 2 is to be connected to the lower layer protocols in the simulated node, typically a DLL UMI. If the payload of the Ethernet frame is not related to ARP, then it is passed to the upper layers as an IP packet, by outputting to the outlink 2.

Connection Map — Outlinks

1 (`ethernetframemsg`)**:**

Outlink 1 is to be connected to the lower layer protocols in the simulated node, typically a DLL UMI. Any Ethernet frames to be sent out to the simulated network are output on this outlink.

2 (`ippacketmsg`)**:**

Outlink 2 is to be connected to the upper layer protocols in the simulated node, typically a Router UMI. The IP packets that are destined to the simulated node, and that are not related to ARP, are output on this outlink.

DLL UM Component

Description

Instances of DLL UM components convert the Ethernet frame they receive from upper layers into a series of bits to be passed to the physical layer simulator. Upon reception, they check if the destination MAC address is the MAC address of their simulated node or the broadcast address, and pass or block the frame accordingly.

Message Types

Ethernet Frame Message (`ethernetframemsg`)**:**
An Ethernet frame message contains an Ethernet frame.

Physical Signal Message (`physignalmsg`)**:**
A physical signal message contains a series of bits, which are meant to be the output of the data link layer to the physical layer.

Connection Map — Inlinks

0:
Inlink 0 is reserved for configuration messages for the DLL UMI.

1 (`ethernetframemsg`)**:**
Inlink 1 is to be connected to the upper layer protocols in the simulated node, typically an ARP UMI. When an `ethernetframemsg` is received, the corresponding physical signal is output to outlink 1.

2 (`physignalmsg`)**:**
Inlink 2 is to be connected to the physical layer simulation. When a `physignalmsg`, an Ethernet frame is constructed from the signal received, the MAC address is checked to see if it is the MAC address of the simulated node the DLL UMI is in, or if it is the broadcast MAC address, then the Ethernet frame is passed on to upper layers by outputting on outlink 2.

Connection Map — Outlinks

1 (`physignalmsg`)**:**
Outlink 1 is to be connected to the physical layer simulator. The bits output to the physical layer are sent out to this link.

2 (`ethernetframemsg`)**:**
Outlink 2 is to be connected to the upper layer protocols in the simulated node, typically an ARP UMI. The Ethernet frames received from the physical layer with destination address of the simulated node the DLL UMI is in, or that are broadcasted, are sent out to this link.

Physical Simulator UM Component

Description

Instances of Physical Simulator UM components simulate the physical layer in the network. They distribute the transmitted signals, be it on a wire or over wireless, to the receivers that receive the signals.

Message Types

Physical Signal Message (`physignalmsg`)**:**
A physical signal message contains a series of bits transmitted by a transmitter.

Connection Map — Inlinks

0:
Inlink 0 is reserved for configuration messages for the physical simulator UMI.

$i|i > 0$ (`physignalmsg`)**:**
Each inlink i is connected to a transmitter that produces messages of type `physignalmsg`. Typically this inlink is connected to a DLL UMI. In the case of a transceiver, inlink i and outlink i is used as a pair that is connected to the transmit and receive links of the transceiver.

Connection Map — Outlinks

$i|i > 0$ (`physignalmsg`)**:**
Each outlink i is connected to a receiver that receives messages of type `physignalmsg`. Typically this outlink is connected to a DLL UMI. In the case of a transceiver, inlink i and outlink i is used as a pair that is connected to the transmit and receive links of the transceiver.

Select Hub UM Component

Description

Instances of Select Hub UM components serve the other component instances who wish to wait for events about file descriptors. They may or may not work in a blocking manner, depending on the timeout value which indicates the maximum time they might block waiting for events.

Upon checking the file descriptors for any events, a Select Hub UMI sends a message to an outlink which is supposed to be connected to one of its inlinks. The reception of this self message causes a new check on the file descriptors. This effectively implements a continuous loop, while at the same time allows the component platform to activate other UMIs for them to process their messages.

Instantiation Constraints

A Select Hub UMI would typically be a singleton in its EU. Alternative designs are not impossible, but hard to manage.

Checking events in file descriptors when the self message is received, might result in, depending on the activation policy of the EU controller, a direct relationship between the number of messages waiting in the reception queues of the instances and the interval between two checks on the file descriptors.

Message Types

Select Hub Control Message (`selecthubcontrolmsg`)**:**
Select hub control message tells the Select Hub UMI to start or stop watching for events on the file descriptors, and the timeout value for determining at most how much time can be spent blocked waiting for events. They are also used by a Select Hub UMI to communicate that it has stopped, possibly due to an error.

Select Hub Self Message (`selecthubselfmsg`)**:**
The self message indicates the Select Hub UMI that it is time to check the file descriptors for events again.

Select Hub File Descriptor State Message (`shfdstatemsg`)**:**
A message of this type contains the request to start or stop watching for a file descriptor for events that signify that data is available for reading, that there is room is available for writing, or that an exception have happened.

Select Hub File Descriptor Event Message (`shfdeventmsg`)**:**
A message of this type informs that one or more events that have been waited for in a file descriptor being watched, due to a previously received `shfdstatemsg`, have happened. It includes the type of the events, as well as the file descriptor.

Connection Map — Inlinks

0 (`selecthubcontrolmsg`)**:**
Inlink 0 is connected to the instance who will start, stop, or configure the Select Hub UMI. This is typically the first CUI.

1 (`selecthubselfmsg`)**:**
Inlink 1 is connected to outlink 1 of the same Select Hub UMI. This is used for creating a loop that does not block processing in the whole EU.

$i|i > 1$ (`shfdstatemsg`)**:**
Each inlink i is connected to a UMI that wants to watch for events in one or more file descriptors. When an event happens in a file descriptor which have been watched due to an `shfdstatemsg` received from inlink i, an `shfdeventmsg` is produced on outlink i.

Connection Map — Outlinks

0 (`selecthubcontrolmsg`)**:**
Outlink 0 is connected to the instance who would want to be informed if Select Hub UMI stops as a result of an error.

1 (`selecthubselfmsg`)**:**
Outlink 1 is connected to inlink 1 of the same Select Hub UMI. This is used for creating a loop that does not block processing in the whole EU.

$i|i > 1$ (`shfdeventmsg`)**:**
Each outlink i is connected to a UMI that wants to receive events being watched for on some file descriptors. When an event happens in a file descriptor which have been watched due to an `shfdstatemsg` received from inlink i, an `shfdeventmsg` is produced on outlink i.

Index