THE DESIGN OF XML-BASED MODEL AND EXPERIMENT DESCRIPTION LANGUAGES FOR NETWORK SIMULATION

by

Andrew W. Hallagan

A Thesis

Presented to the Faculty of

Bucknell University

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science with Honors in Computer Science

June 1, 2011

Approved:

L. Felipe Perrone
Thesis Advisor

Stephen Guattery
Chair, Department of Computer Science

Acknowledgments

Long after I have forgotten the details of regular tree grammars and XML parsing techniques, I will remember my mentor and friend, Felipe Perrone. He taught me a little bit about computer science and a lot about researching with my head up and learning how to learn. I would also like to thank Peg Cronin, who was an enormous source of support during this process. She understood the important stuff, like if we can't take a "plunge" we should at least have our "realm"! I am sorry that it is only just now I have found such a wonderful ally and friend. I also appreciate the time spent by my faculty readers, Sharon Garthewaite and Shane Markstrum. Their comments have helped polish this work into something scholarly. I would also mention Bryan Ward, who for years has been a great collaborator, co-conspirator, and comrade. Inspiration doesn't happen in a vacuum and Bryan has been a friend more inspiring than he probably realizes. Finally, my family has seen all of this and everything else from the beginning. Even if they don't understand the next 85-or-so pages, they'll probably read most of them anyway, which makes me smile.

Contents

List of Tables

List of Figures

Code Listings

Abstract

Empirical study of complex computer networks is costly, disruptive, and simply impractical. Simulating network models is a viable alternative, though this too has problems; the process of effectively incorporating network simulation results into a larger experimental study introduces many potential sources for error, which are known to undermine the credibility of published work. The conspicuous absence of such fundamental information as the simulator engine used, the size of the simulation, details of the PRNGs used, etc., and poor use of statistical methodology are examples of procedural errors that have often made the replication and scientific validation of network simulation studies nearly impossible.

The Simulation Automation Framework for Experiments (SAFE) is a project created to raise the level of abstraction in network simulation tools and thereby address issues that undermine credibility. SAFE incorporates best practices in network simulation to automate the experimental process and to guide users in the development of sound scientific studies using the popular *ns-3* network simulator. This thesis presents my contributions to the SAFE project: the design of two XML-based languages called NEDL (*ns-3* Experiment Description Language) and NSTL (*ns-3* Script Templating Language), which facilitate the description of experiments and network simulation models, respectively. The languages provide a foundation for the construction of better interfaces between the user and the *ns-3* simulator. They also provide input to a mechanism which automates the execution of network simulation experiments. Additionally, this thesis demonstrates that one can develop tools to generate *ns-3* scripts in Python or C++ automatically from NSTL model descriptions.

Chapter 1

Introduction and Background

Computer networks leverage the power of many, and they have become more powerful than ever for a few reasons: the cost of microprocessors continues to fall ever lower, the computational power of these processors continues to rise, and mobile devices using these processors are becoming ubiquitous. Researchers, engineers, government leaders and business people seek to leverage the power of computer networks in many different applications from disease control to energy consumption to military combat. We can classify computer networks into three sets: hardwired, wireless, and ad-hoc. The members of these networks are individual machines, perhaps laptop and desktop computers. They could also be hand-held smart phones or even autonomous devices that operate without direct user input. These members are called *nodes*. Nodes in a network can send messages to one another and typically do so by first breaking the message up into small, discrete chunks called *packets* and then transmitting a stream of these packets to the receiving node. There are many advantages to this method, one being that packets can move independently of one another which means they need not all take the same route from sender to receiver. Additionally, packets are often lost along the way because of excessive network traffic, busy servers or unplugged cables, but because the size of the packet relative to the entire message is small, the cost of re-transmitting the dropped packet is minimal.

The nodes in a hardwired network are connected with wires, such as Ethernet cable, and packets are sent back and forth along this cable according to a fixed protocol. Computers that contain the right hardware and software to follow this protocol can communicate with each other and with servers connected to the Internet. Wireless networks work in much the same way; the medium of communication in

this case is electromagnetic radio waves, and the communication protocol might be the IEEE 802.11 "WiFi" standard. Wireless ad-hoc networks are a third type of network in which nodes can freely communicate with each other while moving about arbitrarily. The idea is that the path of communication between any two nodes changes with the network topology, and members of these networks are in implicit agreement that messages between a sending and receiving node may in fact be routed through other nodes along the way, consuming a certain computational resources of these intermediate nodes.

There are innumerable variations to the kinds of computer networks used in research, government, or industry. They each exhibit different behaviors in regards to properties like speed, bandwidth, security vulnerabilities, power consumption and mobility, among others. Scientists find these behaviors intriguing, and worth studying.

My work addresses the ways in which these networks are studied, and it is helpful to begin from the beginning: the scientific method and study of systems. In formal terms, a *system* is the name we give to something under study. Averill M. Law, author of *Simulation Modeling & Analysis* [29] defines a system as "a collection of entities . . . that act and interact together toward the accomplishment of some logical end." Of course, this definition is made intentionally broad, and researchers talk about a "system" in many different ways depending on the context. My work is generally concerned with computer networks, and so here the systems in question are comprised of computers communicating with each other using various pieces of hardware and software, cables, and radio transmitters. These systems also include the information flowing through the network and the protocols of communication, along with their physical location and surroundings.

Scientists have many ways to study a system, and these methods are mapped out in Figure 1.1 (source: Law [29]). The goal with each of these methods is to answer questions about a system in order to learn more about its behavior. The most direct method for doing this is by experimenting with the actual system; changing it physically and observing its operation under new conditions. For many systems, including computer networks, this is often too costly or too disruptive to do [30]. Consider a corporation wishing to test its network's response to various kinds of power failures. Actually bringing down parts of the network would interfere with day-to-day operations and pose an unacceptable disruption to business. To avoid this, researchers often experiment with a *model* of the system. Models can either be a physical replica of the system or a mathematical representation, but all of the models discussed here are of the mathematical kind (and hereafter I will simply use

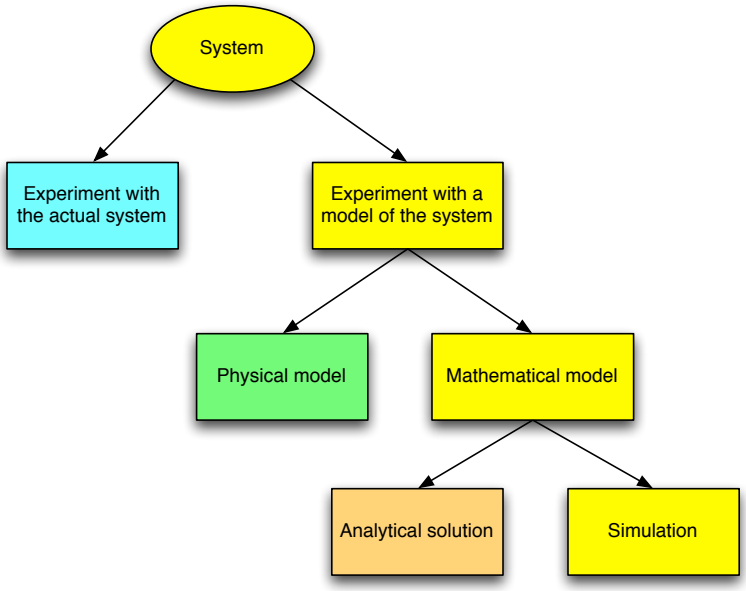

Figure 1.1: Ways to study a system.

"model" as shorthand for "mathematical model"). A mathematical model is a set of logical and quantitative relationships developed to represent a system. The equation $d = v \cdot t$ is a mathematical model relating the distance d an object travels to it's velocity v and the time t spent traveling. Researchers can ask certain questions of a system and use a model to predict the system's would-be response [29]; in this simplistic example we can predict the distance an object travels if we know the values for v and t.

If a model is simple enough, a researcher may be able to answer their questions directly using mathematical techniques such as algebra, calculus, or probability theory. In the $d = v \cdot t$ model, we can solve for any variable given the other two using simple algebra. The two variables whose values are given are called the *model inputs,* and the third variable we solve for is called the *model output* or *model response.* The answer produced in this case is called an *analytical* solution. Many real-world systems are exceedingly complex however, and are only represented accurately with exceedingly complex models. It is not feasible to answer questions about these models analytically; the mathematical techniques needed are either very difficult or impossible. To overcome this, researchers use *simulation.* In a simulation, we provide the input values and evaluate a model numerically, usually using a computer. By simulating enough model inputs, a researcher can begin to construct a *metamodel* – a model of a model

– which most often manifests itself as a regression equation tying model response to one or more model inputs. This metamodel is used to infer the kinds of mathematical relationships among components within the model that we cannot derive analytically. These mathematical relationships point to real relationships that could potentially exist within the system, depending on the accuracy on the model [29].

Computer networks, especially large-scale computer networks, fall within this class of exceedingly complex systems. Some can contain hundreds of millions of nodes, comprising both static and dynamic structures [10]. Simplistic analytic models of these networks are useful for a coarse level of analysis but are limited in the problems they can tractably solve. Simulation of more complex network models provides a feasible way to understand network behavior and optimize network designs before physical deployment [35].

There are many compelling reasons for studying networks of this complexity and size. In their white paper on the U.S. Department of Energy's efforts to develop a better understanding of complex networked systems, Brase and Brown [10] outline many important applications for network simulation research, many of them related to operations of the U.S. federal government. Broadly speaking, the U.S. government has an important stake in knowing more about the inefficiencies, vulnerabilities and design flaws in its own networks. For example, developing the capability to simulate the nation's power grid at scale is a crucial step on the way towards expanding and modernizing its operations [10]. The Department of Energy also relies on the ability of scientific groups to reliably and securely share significant amounts of information between themselves over networks around the world in order to promote further scientific discovery [10]. Other kinds of networks of interest to the government include social networks, biological networks and networks that describe the spread of disease. An improved understanding of all of these will have a substantial impact on federal scientific missions and public policy.

The argument made by Brase and Brown [10] is that the U.S. Department of Energy is in a position to develop this knowledge through further network simulation research, but has not yet done so. They write that

> [The Department of Energy] must continue the development of next-generation complex networked systems that are more secure, less brittle to unexpected events, and more controllable. For these emerging efforts to be successful, it is essential that a firm intellectual foundation be provided for understanding and simulating large-scale networks.

Creating this foundation requires more substantial study into network simulation techniques. In this arena of complexity however, such study is not as straight-forward or intuitive as researchers might assume. Ultimately, my work addresses certain obstacles that impede the production of credible research into these complex systems. Such obstacles have their root in the modeling process described previously, as well as in the experimental process that uses those models. There is much academic literature on the subject of experimental design [29, 32, 46], and the material is important enough to warrant explicit discussion here. Below, I give a more rigorous description of the experimental process and experimental design, especially as it relates to simulation. This material is central to the rest of my own work.

In general, an experiment involves changing the settings of *factors* in a model. Law [29] defines a factor as "the input parameters and structural assumptions composing a model." Factors are sometimes are sometimes called *variables* in a simulation. For instance, suppose we have a model with input parameters `SPEED` and `DISTANCE`. A researcher may wish to determine how changes in these inputs affects the model's output behavior. The decision as to which factors should remain fixed and which should be varied in an experiment is a choice left up to the experimenter, and depends on the goals of the experiment rather than the model itself [29].

A *design*, as defined by Sanchez [46], is a matrix which has a column for each factor being varied in the experiment. The entries in each column are the settings for that factor – the *level* it will take on for any particular simulation. Thus, each row represents a particular combination of factor levels, and is known as a *design point* (see Table 1.1). Given a design point, we can plug in the given level values corresponding to each factor and run the simulation. The data collected during that simulation run is called the *response*.

SPEED	DISTANCE
0	0
1	1
1	2

Table 1.1: A Simple Design

Of course, it would be difficult to draw conclusive results from the experiment in Table 1.1 because the change in response between the first design point and the second could have been the result of the change in `SPEED` or the change in `DISTANCE`, or both. In this case, any change in model behavior cannot be conclusively explained

without testing more design points. As it is now, the effects of changing `SPEED` and `DISTANCE` are *confounded*.

The end goal of any simulation study is to determine the effect certain factors have on the behavior of a model. If the experimenter has absolutely no idea where to begin, a 2^k*factorial design* is often chosen [29]. In this scheme, each factor is assigned two levels, chosen far enough apart that they will cause observable differences in model behavior, but close enough that they do not produce wildly different results. If we wish to measure every possible configuration of a model with k factors being varied, then the total number of design points will be 2^k, which is where this scheme gets its name.

Of course, even with just a handful of experimental factors the number of design points can quickly spiral out of control. If the number of factors stays constant then increasing the number of levels each will take on increases the number of design points polynomially. If the number of factors in the experiment increases, the number of design points increases exponentially.

Since computer networks are time-dependent systems, simulation of a particular design point often ends up requiring that a model be evaluated at many successive points in time. Additionally, network models are usually built to exhibit certain kinds of random behavior that reflect real-world variation (e.g., the amount of network traffic between a user's personal computer and a web server). In this case, researchers must simulate each design point multiple times to estimate a model's response within reasonable statistical confidence intervals. For these reasons, network simulation is often computationally expensive and time-consuming. Law [29] explains that a major goal of experimental design in simulation is to identify which factors have the greatest effect on model response, and to make that identification with the least amount of simulating possible. This process is called *factor screening* or *sensitivity analysis*. After learning more about which factors really do matter in changing response, an experimenter can begin to construct the kind of metamodel mentioned previously. The metamodel allows researchers to make judicious decisions about which additional design points are worth spending time to run in simulation, and which can be predicted using their metamodel.

Simulation itself begins by using software to define some network model and observing how it behaves. This definition is written formally in a computer language and is called the *model description*. Every simulation requires a model description, which contains information about the number of nodes involved, the way those nodes are moving about, how much bandwidth is being used, the number of Internet access

points, etc. The model description is subsequently given to a *simulator engine* (sometimes just "simulator" for short), which is a program that executes the simulation and reports the results of an experiment.

Researchers have a number of network simulators at their disposal, including *ns-3*, *ns-2*, *GloMoSim*, *OPNET*, *CSIM* and others [27]. Not only are the models difficult to describe, but each simulator engine uses a different language for model description, which makes it very difficult for researchers to share and reproduce their work across platforms. The differences in model description languages, compounded by the sheer complexity of the models being described have had a serious negative impact on the credibility of many past simulation studies. Researchers should be able to trust the published results of their peers and have confidence that their own results are experimentally sound. But, in their report on the state of network simulation credibility, Kurkowski et al. [27] identified a number of published studies where this was not the case. The authors explain that without exhaustively listing all of the parameters used in a simulation study, researchers make it very difficult for others to replicate their work — a crucial piece of the scientific method. This sort of credibility gap in published work is detrimental to the building of the "intellectual foundation" in network simulation that Brase and Brown [10] speak of.

As it is today, if researchers want to reproduce their colleagues' findings (obtained using a simulator different from their own), they must translate a model description file by hand, from one language to another. This creates enormous room for error. There is the potential for making careless translation mistakes, and also the possibility of not being given the original model description in its entirety; certain parameters may be left unspecified in the original description, with the intent that the underlying simulator will provide some default value instead. If these default values are not explicitly made clear in the original model description, the second researcher is forced to either guess what they were, or let their own simulator choose a default value (which may or may not be the same one). It would seem that these mistakes could be easily avoided in computer simulation; given that in simulation studies it is possible to isolate the effects of *all* external forces, research conducted using computer simulation should allow for exact reproducibility [50]. Nevertheless, Kurkowski et al. [27] identified simulation setup as the most often ignored experimental phase in mobile ad-hoc network simulation studies (a subset of all network simulation studies). This undermines the scientific credibility of a study from the beginning.

While Perrone et al. [41] support the continuing emergence of new tools and systems for network simulation, they argue that a concerted effort towards increasing the *credibility* of research produced with these tools is requisite before any reliable

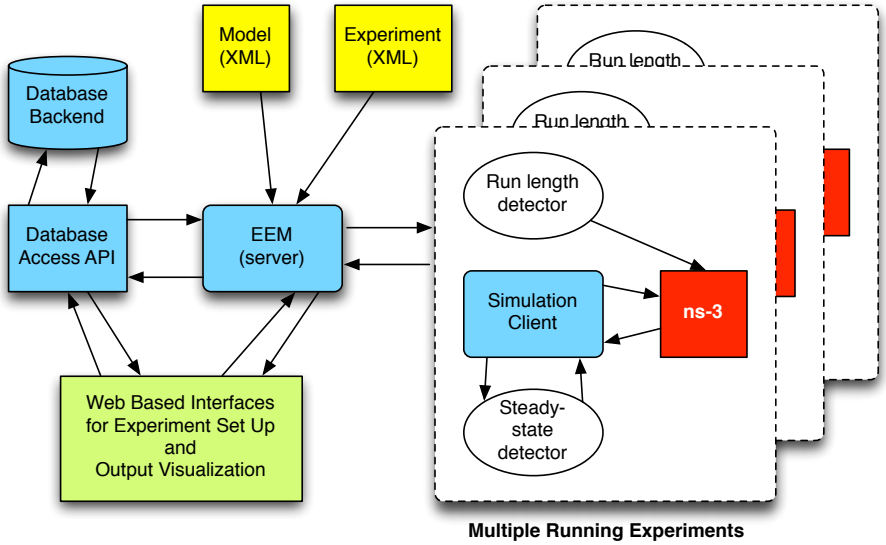

Figure 1.2: A high-level view of the SAFE architecture.

progress can be made. A central claim of [40] and [41] is that this credibility will come when simulators can provide complete automation of the experimental process. In my honors thesis work I have made significant steps towards the realization of this goal by developing ways to create composable network simulation models for the *ns-3* simulator that can be validated and transformed with Extensible Markup Language (XML) technologies.

This project exists in the larger context of Dr. Felipe Perrone's work on the *ns-3* network simulator. It is part of a deliberate effort by Dr. Perrone and others to provide tools that can substantially increase the credibility of network simulation studies. He and other members of the *ns-3* development team at the University of Washington and the Georgia Institute of Technology have secured a four-year N.S.F. grant (CNS-0958142) in order to fund the development of software frameworks to automate the experimental process within *ns-3*. Dr. Perrone and Bucknell's contribution to this work is the *Simulation Automation Framework for Experiments* (SAFE), whose high-level architecture is presented in Figure 1.2. The following description of SAFE, copied literally from the project's documentation, provides an overview of the system's goals and general structure:

> "The automation framework will consist of user interfaces, description languages, and tools that will help users of varying levels of expertise to

> produce more credible simulation experiments with *ns-3*. The functionality offered will enable the user to define, deploy, and control *ns-3* simulation experiments that are methodologically valid and easy to reproduce by third parties. The framework will include tools for: model composition; structural validation of the model; configuration of model components; description, deployment, and control of experiments; output data processing and storage; and reporting of experimental setup. Although the framework will offer graphical user interfaces, more experienced users will be able to access automation functionality via the command-line."

My work lies in the areas of experiment description, model description and automatic code generation. I have developed the *ns-3 Experiment Description Language* (NEDL), the *ns-3 Script Templating Language* (NSTL), and a set of tools to support their use with the *ns-3* simulator. The end goal is to provide an automation framework that begins with experiment and model description and ends with automatic generation of *ns-3* simulation script code. My thesis is thus:

The community of network simulation researchers stands to benefit from a substantial increase in experimental credibility. The development of languages and tools to support automated experiment description and model description help increase that credibility. The implementation itself and especially the research I have conducted in this work will impact the simulation experiment process on the ns-3 *simulator in a positive way.*

This document describes the conceptual design of the NEDL and NSTL languages and tools, as well the specific technologies used to create them. I have implemented these languages using the Extensible Markup Language (XML) specification. Since so much of this work centers on XML technology, it is appropriate in this introduction to first provide some background on XML itself.

XML is a markup language standardized by the World Wide Web Consortium (W3C) [6], an "international community that develops standards to ensure the long-term growth of the Web." In general, a markup language uses some notion of *markers* that are present inside what would otherwise be unstructured textual data. The markers are used to facilitate the procesing of information. The HyperText Markup Language (HTML) is one markup language most laypeople have come into contact with, thanks to its ubiquity on the Web. HTML is used to mark pieces of a web page so that it can be automatically interpreted by a web browser and rendered (basically) the same way to anyone who visits the page, regardless of their particular machine,

operating system or web browser.

Just like an HTML document, an XML document is simply a text file that uses a system of markers to define *elements* in a document. These markers are called *tags*, and are identified using angle brackets (`<, >`), which, along with the ampersand (`&`), are the only reserved symbols of the XML specification. The specification allows users to define their own tags and what kind of contents a particular element can contain. A rigorous discussion of the XML specification is beyond the scope of this work; much more detailed information can be found in a number of places on the Web, including the official W3C Recommendation [11]. Nevertheless, for those wholly unfamiliar, the fundamental pieces worth explaining are:

- An XML start tag is just a word surrounded by angle brackets. E.g., `<name>`.
- Every opening tag must have a closing tag, which is identified with a slash before the tag name. E.g., `</name>`. The opening-closing pair together with whatever is contained between them constitute an *element*.
- Between the opening and closing tags, there can exist string content or other XML elements.
- If there is *nothing* between two corresponding opening and closing tags, the tags may be merged into one such that `<name></name>` becomes `<name/>`.
- An element may contain *attributes*, which are key-value pairs inserted after the opening tag name, but before the right angle bracket.

See Figure 1.3 for a diagram of a sample XML element with labels for each of its components.

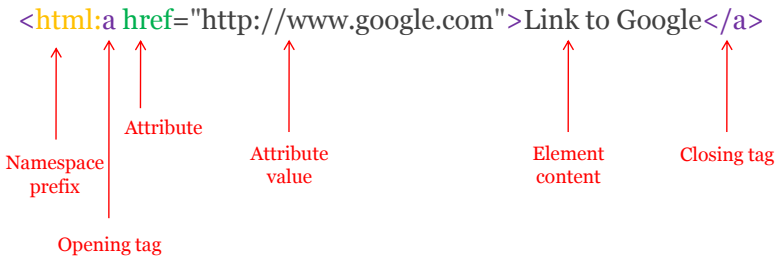

Figure 1.3: An HTML "anchor" element and all of its components.

Following these rules, it is possible to create custom tags that express complex relationships between entities. A small example is shown below. This made-up example is meant to resemble the specification of a particular application-layer model we can call the "Ping Application."

```
1 <application>
2     <name>PingApplication</name>
3     <interval>1.5 seconds</interval>
4     <duration>0.5 seconds</duration>
5 </aplication>
```

Even for someone unfamiliar with simulation models, it should be at least plausible that the hierarchical structure of XML lends itself quite naturally to describing relationships between nested components of a simulation model. The W3C has published specifications for other technologies that can be used for analyzing and manipulating these XML documents.

An original goal of my work was to develop a simulator-agnostic model description language which allowed for model composability as well as automatic translation into various simulator-specific renderings, the first being *ns-3*'s C++. This was an ambitious goal that I found to be infeasible within the time available. NSTL allows for an alternative solution using script templates tailored specifically to *ns-3*. This represents a scaling back of the original goals of my work and is discussed in greater detail in Chapter 7.

Chapter Summary

The overarching goal of this work is to investigate the experiment and model description processes used in network simulation research and find ways to automate this. I have researched these processes and what others are doing to address the same class of problems. The results of that have motivated the particular design choices I have made in the implementation of the experiment description and model description languages, as well as the tools that automate the entire process from experiment and model description to *ns-3* code generation. This work focuses exclusively on the experiment design and execution process on the *ns-3* simulator.

As part of this work, I have developed the *ns-3* Experiment Description Language (NEDL), and the *ns-3* Script Templating Language (NSTL). Because of the enormous complexities involved in transforming abstract model descriptions (in any form, including XML) into *ns-3*-executable C++ models, I have decided to implement NSTL as a language used for developing *templates* of *ns-3* simulation scripts as opposed to a language used to develop standalone simulation models.

Chapter 2

Related Work

The original XML specification was developed in 1996, which means researchers, developers and businesspeople have had 15 years to put XML through a spectrum of different tests and use cases. Because of this, its use has proliferated in many different problem domains, including systems biology [26], business applications [23] and of course, network simulation [7] [44] [45]. XML has been investigated by a number of researchers interested in creating simulations that are universal and portable. It is an appealing language because it is complemented by a suite of W3C-endorsed and third-party technologies for editing, querying, validating and transforming XML documents; it thus is a convenient data exchange format. The standard has established itself as one that is robust, flexible and mature.

The rest of this chapter will focus on work others have done in the intersection of the XML and network simulation domains. The discussion is meant to be a general overview of this sort of work and I will try to cover the ideas presented in broad strokes only. When talking of work whose details do have substantial overlap with mine, however, I will be talking about those specifics too.

2.1 XML, Simulation and Model Description

Rioux et al. [44] present a technology-independent conceptual framework for describing simulation scenarios, validating those scenarios, and then converting those scenar-

ios into software objects that can be used by a particular simulator engine. They show that XML and some of its associated tools (JAXB, XQuery, XSLT and Native XML Database) can be used to implement this framework. Because these tools are available today, the time spent developing such an implementation is greatly reduced. Their work does not match the goals of mine exactly, but their arguments for using XML bolster the decisions made in my own work. They emphasize that in their case, XML was chosen because of the ability to exploit the same technology throughout their entire data pipeline. This includes visual representation, data binding, and stream transformation. They also conclude that although other data encoding formats are available, like the Hierarchical Data Format (HDF) or Network Common Data Form (NetCDF), these binary formats do not offer the same flexibility and ease of use as XML.

Amoretti et al. [7] have developed the Discrete Event Universal Simulator (DEUS) which requires the network topology information to be written in an XML-based language of their own design. The researchers have defined a transformation that maps every topology descriptor in DEUS to an equivalent XML representation, and back again. Configuration information for DEUS is also contained in an XML document. They believe that using XML instead of another proprietary format eases the novice's entry into the project and makes editing the topology and configuration setup much easier, especially when that data must be prepared from some other external source, such as a database. Guiding the novice in their use of the *ns-3* simulator is part of the mission of SAFE and the work of Amoretti et al. [7] supports my claim that XML is well-suited for this purpose.

Röhl and Uhrmacher [45] discuss XML-based model components and show that the declarative specification of XML provides easier database integration, user readability and development of graphical user interfaces. In addition, XML happens to be an excellent data exchange format, allowing researchers to import and export model definitions in a more platform-independent way. A disadvantage of describing simulation models in XML is the problem of converting a declarative specification (XML) to an imperative one (such as an *ns-3* C++ script). For this reason, Röhl and Uhrmacher [45] emphasize that it is important to maintain a separation of code describing the *composition* of a model and code describing the *execution* of that model in the simulator engine. The authors demonstrate an implementation of this framework as part of the James II simulation system. I encountered this same difficulty in trying to separate model composition and execution when creating the NSTL language and code generator module. As I'll discuss in greater detail in Chapter 4 on NEDL and NSTL syntax and Chapter 7 on code generation, I settled on an approach that is not as ideal as the kind Röhl and Uhrmacher [45] advocate.

In their paper on automated analysis of model specifications, Olson et al. [39] describe their work in developing an XML-based version of the Condition Specification (CS) model specification form. Although the particulars of their work do not align exactly with mine, they do discuss their translation process from this XML-based Condition Specification (CS-XML) language into fully-executable C/C++ code. They identify many of the benefits in using XML, including: "the semantic power of XML due to its 'extensible' nature; flexibility of use over other document formats such as binary, fixed-length or delimited text data; portability of XML documents across diverse systems and platforms." XML alone does *not* carry any semantic information, but perhaps the authors were referring to XML's ability to represent many different kinds abstract objects, which can be given semantic meaning by a Schema or parser. Unfortunately, the authors do not produce anything more than a simple proof-of-concept code generation scheme, so we cannot leverage any of their work in that regard. The authors suggest that XSLT may be an acceptable solution for transforming the CS-XML documents into C++ code. I attempted to implement such a solution for SAFE but found XSLT tedious and cumbersome to deal with. The code generation aspects of my work is discussed fully in Chapter 7.

2.2 XML, Simulation and Experiment Description

In regards to experiment description, the development of an XML-based language for this purpose has been a goal in other problem domains besides network simulation. For example, SEDML is an experiment description language used in computational systems biology to describe simulations [26]. Köhn and Novère [26] launched the Minimum Information About a Simulation Experiment (MAISE) project in 2007 to provide guidelines on what sorts of information should be provided to give a full description of simulation experiments. SEDML is the implementation of those guidelines and is a language independent of both the format used to encode models as well as the tools used to execute simulations. SEDML is similar in spirit to the NEDL language I have developed, though it exists in a domain that is far enough away from network simulation that I deemed direct use of SEDML to be too difficult to pursue and instead chose to implement NEDL from scratch.

In the field of network simulation, Canonico et al. [13] propose an XML language for describing network simulation scenarios. Just as I do in my project, the authors construct an XML Schema which defines all valid simulation scenario descriptions. They also defined a set of translation rules using the Extensible Stylesheet Language

Transformations language (XSLT) [15] for translating a simulation scenario descriptions into an executable script for the *ns-2* network simulator. Their arguments are in support of the design decisions I have made while developing NEDL and NSTL, specifically my choice to keep element and attribute structures simple and straightforward to ease the development of front-end GUI modules in the future. Simple XML element structures favor automatic generation and Canonico et al. [13] say outright that "simulation of large scale networks cannot rely on manual generation of the XML simulation scenarios." On the surface, the work appears to have much overlap with my own, but the substantial differences between the *ns-2* and *ns-3* do much shrink this intersecting area.

Bertoli et al. [9] provide a corroborating discussion on the state of network simulation research, acknowledging the usefulness of simulation while cautioning that simulation can sometimes fail or produce non-accurate results. Common causes of such failures and inaccuracies include incorrect statistical techniques implemented in the simulator engine itself and users' mistakes, such as inadequate level of detail adopted to describe the target system, too short simulation time and errors in input parameter values. It is precisely these sorts of errors that the SAFE architecture aims to minimize, if not remove completely. In their architecture of JSIM, the simulation module of the Java Modeling Tools (JMT) suite, the authors use XML documents to provide a separation of what they describe as the "presentation" and "computational" layers of the module. This XML data layer is used to invoke the simulator engine as well as save user-specified models. The graphical user interfaces of JSIM; JSIM*wiz* and JSIM*graph* are also built on an XML layer that guides users through model parametrization and topology layout. The authors explain that an XML-based interface between users and the simulator simplifies the implementation of graphical user interfaces, a long-term goal of interest to the *ns-3* community as well. These interfaces would do much to guide novice users and ease their entry into the system.

2.3 Alternative Model Description Languages

As mentioned previously, the goal in my work of providing abstract model description capabilities to the SAFE framework has been set aside in favor of a simpler, near-term solution: providing *ns-3* script templating functionality. Still, other model description languages do exist, and a discussion of the most prominent ones and why they were not incorporated into SAFE is appropriate here.

The Domain Modeling Language [3] is an alternative model description language that has been used in network simulation before, especially in projects using the Scalable Simulation Framework (SSF) [5]. Perrone and Nicol [42] describe the language as a "simple, but powerful" way to describe models. Like XML, the DML language is tree-based; it uses a set of key-value pairs to describe model entities, where the keys are any alphanumeric string and the values are either strings themselves or other DML entities enclosed in square brackets. The DML Reference Manual [3] contains the following example of valid DML given in Listing 2.1.

```
portmaster [
  _schema ".dictionary.schemas.portmaster"
  serialnum 12345
  hostname "far.near.net"
  tty [ name tty01 speed 40kBs]
  tty [ name tty02 speed 800Mbs]
  tty [ name tty03 speed 4kBs]
  machinespecs [
    _schema ".dictionary.schemas.machinespecs"
    powerconsumption 1600w
    mtbf "20 days"
    replacementcost "$500,000"
  ]
]
```

Listing 2.1: An example fragment of DML.

In this example, the keys that begin with an underscore character are special reserved keywords of the language. Note the keys `tty` and `machinespecs` within the `portmaster` element; the corresponding values for these keys are further DML fragments rather than just simple strings. In this way, DML describes tree structures in much the same way as XML does. The element and attribute names of XML correspond to the key strings in DML, and the contents of those elements and attributes correspond to the value entities in DML. A "simple type" element or attribute in XML is equivalent to a pair of key-value strings in DML, and a "complex type" element in XML is equivalent to a string-DML pair in DML (see [47] for details on simple and complex element types in XML).

DML does have features, however, that XML lacks. Plainly, it is much more concise. The verbosity of XML is a perennial complaint made in a great deal of

literature on the subject [44] [18] [19]. Having a matching closing tag with each opening tag creates redundant information and bloats XML file sizes considerably. DML also supports inheritance models that XML does not. As a markup language, XML of course provides no mechanism to support *types* and thus has no concept of inheritance. Certain schema languages, however, like XML Schema [49], do support something similar to inheritance. XML Schema allows for element "derivation" by extension and restriction, but its utility is limited. DML, as a *modeling* language, does this more naturally, supporting object-oriented style inheritance, and even multiple inheritance.

Related to DML is a language developed by Kiddle et al. [25] called ANML, which stands for "ANother Modeling Language." In their work developing ANML, Kiddle et al. [25] investigated a number of model description languages and found that although DML supported hierarchical modelling, reusability, and the construction of large models, it was difficult to express all of the necessary validation rules they wished to incorporate into their models. XML has robust support for validation, with multiple schema languages and validation tools available. The authors decided, however, that description in XML was still more difficult than with DML, and elected to base ANML primarily on the DML grammar, with support for some of the features available in XML.

In my work, the validation tools are a crucial piece of the SAFE framework, and XML is a good language choice because of the availability and reliability of these tools. The language syntax is indeed verbose and does have some limitations for model description, given that it is never anything more than a markup language. However, there is an abundance of tools for parsing and manipulating XML that can side step its limitations in model description. Additionally, its status as a World Wide Web Consortium Recommendation is a benefit too since this makes it easier to support further expansion and modularization of the description process using other web-based technologies. Front-end web interfaces or GUIs can easily generate XML automatically and exchange data with web-based systems much more efficiently than would be possible with a language like ANML.

2.4 XML, Simulation and Ontologies

Taken broadly, the mission of the SAFE framework falls within the realm of ontology development for the network simulation community. Though ontology development

is outside the boundaries of this work, it very well may be pursued in the years ahead for SAFE. It is therefore useful to discuss what ontologies are, their role in network simulation research, and how my work makes positive steps in the direction advocated by researchers in the field.

Benjamin et al. [8] define an ontology as "an inventory of the kinds of entities that exist in a domain, their salient properties, and the salient relationships that can hold between them." The key here is the notion of formalizing knowledge in a such a way that machines can perform work on that knowledge. As described in [20] and [31], ontologies are hoped to be the next milestone in the development of the World Wide Web; they promise to enable the *Semantic Web*, an additional layer to the Web stack that gives meaning to the billions of documents, web pages, databases and files that exist on the Web today. Ontologies can also be used in applications beyond the Web. Researchers in many scientific fields are beginning to organize their knowledge into ontologies; examples include the *Gene Ontology* in Biology, *EngMath* in Engineering Math, *EHEP* in Physics, and *OntoNova* in Chemistry [31].

Benjamin et al. [8] discuss the growing need to develop ontologies for the modeling and simulation communities and the benefits of doing so. The authors speak of developing *ontology libraries:* large knowledge bases that support constant revision and integration of structured, domain-specific ontological information which can then be extracted at many different levels of granularity in application situations. There are important benefits in developing ontologies for the field of modeling and simulation. First, they spur the creation of *knowledge-based systems* that describe not only the makeup of systems and their components, but the semantics behind those relationships and how to use them. Secondly, ontologies form the basis for the development of *reference models* in the modeling and simulation field, which can be re-used in many applications. Ontologies play a critical role in bringing about simulation model interoperability, composition and information exchange at the semantic level.

The link between my work with experiment and model description languages and the work of others towards ontology development is XML. Fishwick and Miller [20] explain that one way an ontology may be encoded is XML, the standard the authors call the "lingua franca" of the Web. The W3C has published standards for a stack of XML-based languages that are to be a part of the Semantic Web initiative. These include RDF (Resource Description Framework), RDF-S (RDF Schema), OWL 2 (Web Ontology Language, version 2), and SWRL (Semantic Web Rule Language). (All of these can be referenced from the W3C website, `http://www.w3.org` [6]).

XML is thus the syntactic layer upon which the Semantic Web is to be built, and

researchers are already attempting to provide the functionality OWL 2 and SWRL aim to provide using various dialects of XML, such as the Mathematics Markup Language (MathML) [14] and the Chemical Markup Language (CML) [34]. My work is well-aligned with this movement since it provides the same ingredients (an XML dialect and associated schema) that more mature language projects have provided. The Simulation Reference Markup Language (SRML) [43] and the Extensible Modeling and Simulation Framework (XMSF)[12] are two examples of this in the field of network simulation. Researchers generally agree that the development of XML-based languages like SRML, XMSF, NEDL, and NSTL provides a useful source of information for the creation of modeling and simulation ontologies [20].

Chapter Summary

The Extensible Markup Language has established itself as a standard that is simple enough to be quickly understood and adopted, and flexible enough to find roles in a range of applications across many different domains. This chapter discussed applications especially close to certain components of SAFE. The XML standard has been endorsed by the World Wide Web Consortium, which has also endorsed an array of other XML-related technologies and standards that support virtually all of the operations a programmer might wish to perform on an XML document. Researchers in the field of network simulation have voiced the need for standardized model and experiment descriptions to better support simulator interoperability, code robustness, ease of simulator maintenance, and simulation credibility. Though ontology development is beyond the scope of this work, providing XML-based languages for experiment and model description is a necessary first step in the process of creating a domain-wide network simulation ontology.

Chapter 3

Conceptual Design

Although the *ns-3* simulator is a powerful system, it can be intimidating for new users because of its complexity. The time it takes students to progress through the codebase of *ns-3* makes it challenging for educators to integrate the simulator into curricular activities. My work has involved designing NEDL and NSTL as languages that can facilitate a higher-level description of an experiment design space and the construction of an *ns-3* script template. As I demonstrate, this helps ensure that only sound scientific research is conducted on *ns-3*. This chapter describes the process of designing and testing these languages, including the specific goals they meet, the reasoning behind having XML-based languages in the first place, and how the well languages fit their intended purpose.

3.1 Roles of NEDL and NSTL

The NEDL language aims to facilitate the description of an experiment space as described in Chapter 1. This includes language constructs that support the listing of experimental factors along with their associated lists of levels. Since not all design points in an experiment space are necessarily worth the time to run in simulation, an important feature built into NEDL is the ability to "prune" design points that are either meaningless, irrelevant or uninteresting *before* the simulation execution and data collection stages.

NSTL is intended to support fast and intuitive construction of *ns-3* script templates. It is designed such that existing *ns-3* scripts can be quickly augmented with special NSTL constructs that support templating at two levels of granularity: basic parameter substitution as well as the swapping of interchangeable code blocks. I developed NSTL for use within the SAFE framework on the *ns-3* simulator, though there is no reason why it could not be used for any application requiring these templating features.

3.2 Language Goals

In order for NEDL and NSTL to provide a meaningful contribution to the *ns-3* architecture and the experimental design process, they both should meet some basic design objectives. The languages must be:

- Expressive: NEDL, in particular, should have the power to describe simple design spaces while providing flexible constructs that allow for more complex manipulation of the experiment design space.
- Compact: XML is verbose by its own nature and both languages should strike a balance between having larger element structures and tag names that provide self-description and smaller ones that favor brevity.
- Intuitive. The XML tag names and element structures should be easy enough to write and edit by hand as well as automatically generated by some other module (i.e., a front-end GUI).

Expressivity is an important feature that merits further discussion. With NEDL, it is important that the language has the ability to describe the most basic experimental design spaces (e.g., a list of experiment factors each with a corresponding list of levels) in a simple, straight-forward way. NEDL should have constructs that allow one to describe more complex experimental design spaces too. These complex design spaces come up in cases when:

- one would like to "prune" some of the factor/level combinations out of the experiment space that are of no interest to them;

- one would like to use a list of parameter values available in some external file, perhaps for more than one factor in their experiment;
- one would like to describe the levels of a factor in some mathematical way, rather than having to enumerate them by hand (e.g., all multiples of 50 between 100 and 1000).

3.3 Description with XML

This work demonstrates the feasibility of using XML-based languages to describe network simulation experiments and models. Currently, researchers using different model description languages must manually translate each others' model descriptions for the purpose of reproducing scientific results. The many different model description languages that each simulator uses, however, leave ample room for translating mistakes. Lack of sound experiment design on the part of the researcher and lack of safeguards built into the simulator to catch these mistakes compounds the problem by introducing inaccurate results and assumptions.

To address this, I argue that the network simulation community should adopt languages into their frameworks that enable higher-level experiment and model descriptions. These descriptions can then be translated into the various dialects that each specific simulator requires. Currently, there are no such languages in the network simulation domain, but XML seems to be an attractive candidate; it is a W3C-endorsed standard that is complemented by a suite of related technologies for editing, querying, validating, and transforming XML documents. The immediate availability of these tools bolsters the decision to adopt an XML-based language in this domain.

A subsequent goal of interest to the network simulation community and certainly *ns-3* in particular, is to address the problem of model validation. Very often, a user will want to implement some amount of customization to the "base" models provided in the simulator library. To conduct a sound scientific study, however, users ought to have a detailed and comprehensive understanding of the underlying simulation engine and how specific model components fit together in that engine [41]. It is not reasonable to expect every user to have this knowledge, much less novices or students. Indeed, in the past four years, nearly 2 million lines of code have been changed in the *ns-3* code base, and the *ns-3* simulator has been easily receiving over five-thousand downloads per month for the past year. But according to the *ns-3*

version 3.9 documentation, there were just 67 contributors involved in the system [4]. The lopsidedness is not surprising for an open-source project like *ns-3*, but it does mean that among the vast number of people who use the system, it is likely only a tiny fraction deeply understand all of its internal pieces.

Thus, a great many users would benefit from the development of experiment and model description languages. I designed languages which can be validated at many levels of granularity. Again, XML was well-suited to this purpose thanks to the many already-defined XML schema languages available, including the W3C-endorsed XML Schema [49], as well as alternative schema languages such as RELAX NG [16] and Schematron [2].

3.4 NEDL and NSTL in Practice

My work designing and implementing NEDL and NSTL brings to light many of the complexities involved in language design, both in a general sense and in regards to the specific problem domain of experiment and model description for network simulation. Experiment description and model description are facilitated in some way by every simulator engine, usually in ways that are tightly connected to the simulator itself. To take those descriptions out of context and de-couple them from the simulator for which they were designed is hard.

Although the grandest vision would have an experiment and model description language that can be translated into every popular simulator's specific dialect, this work narrows the scope considerably by choosing *ns-3* as the simulator engine whose model description language we translate to. Since the *ns-3* simulator is written in the C++ programming language, C++ has become the target language when translating NEDL and NSTL documents. There are many difficulties in doing this, perhaps the most fundamental of which is attempting to represent information encoded in an algorithmic, object-oriented language, using a tag-based markup language. The particular pieces of an *ns-3* script that ultimately define an experiment and what models are used exist within all sorts of other supporting code. All of this code of course, is developed with the knowledge that it is *executable;* it defines an algorithm, a process, a sequence of events. The great difficulty lies in the fact that XML by itself, does not.

Experiment description proved to be the easier task simply because no immediate

knowledge of network simulation itself is even required for one to consider experiment description in XML. There is no shortage of literature on the topic, and as discussed in Chapter 1, Law [29], Morris [32], and Sanchez [46] all give domain-independent accounts of what an experiment *is* and how to go about formally describing one. I decided to limit NEDL to describing experiments whose factor/level combinations were in the end just key-value pairs that could be represented in machine terms as string-string pairs or string-number pairs.

In the SAFE framework, running a simulation means executing a particular script – a static C++ file. An experiment that involves changing the levels of one or more factors requires a new simulation to be run for each design point. (In fact, this is simplified somewhat since a simulation may be run many times over in order to obtain a suitable statistical sample for the metric of interest. But this still only requires one simulation script to be re-run repeatedly.) Thus, any process involving the validation and parsing of NEDL files requires no knowledge of the inner workings of any particular script itself. Furthermore, NEDL was developed with the intention that users should not necessarily need to know the program structure of a particular simulation script in order to design an experiment. Instead, users can focus on looking at experiment factors from an abstract point of view and developing lists of levels to associate with them. These levels are later directly substituted into the static *ns-3* script files before the scripts are run on the simulator.

Developing a model description language was the harder task because models in *ns-3* are built using software classes and "connecting" them together sequentially, using various method calls and data structures. I ultimately decided that trying to encode the structure of these classes and their relationships to one another, as defined statically in the *ns-3* object hierarchy as well as dynamically with the use of references and pointers, was not feasible in the time that I had.

Instead, I elected to use "template-based" model description, which means models are built out of static blocks of C++ code within an *ns-3* script. These blocks make up a script template, which is the reason for naming the language the *ns-3* Script Templating Language. Once a researcher has written and validated their code, they can remove any kind of strings or numbers from the script and insert special NSTL markers instead. The NSTL markers facilitate the substitution of different parameter values into the script as well as swapping interchangeable blocks of code to generate scripts with different model components. The lowest-level markers are used for simple string-substitution and correspond to the factors listed in a NED document. These markers are replaced with values which are defined in the level lists described with NEDL. Thus, my original goal of enabling model validation and composability has

been pared down somewhat. In some ways this provides researchers with a much finer degree of control since they can tweak their simulation scripts by hand and then just insert custom markers after verifying the script executes as intended. This is all discussed in much greater detail in Chapter 7, on code generation.

Chapter Summary

This chapter described the NEDL and NSTL languages from a conceptual standpoint. NEDL and NSTL are languages that enable a higher-level description of experiments and models in *ns-3*. The goals of NEDL are to provide simulator-agnostic experiment descriptions that can be transformed into a set of design points that pair factors represented as strings with corresponding levels, represented as either strings or numbers. This functionality alone is an immense help to researchers using standard factorial experiment design. Those who wish to prune specific design points from their experiment design space can do so with special constructs built into NEDL.

Ideally, high level model descriptions *should* be expressed in a "simulator-agnostic" language. However, given the great complexity in transforming abstract model definitions from a markup language to *ns-3*-compatible ones in an imperative programming language, I opted for a template-based form of model description. In this scheme, models are defined exactly as they would be in an *ns-3* simulation script, but with hard-coded parameter values replaced with special NSTL markers. Entire code blocks are also delimited with NSTL markers too, which facilitates swapping of interchangeable code blocks — an approximation of sub-model composability.

Chapter 4

NEDL and NSTL Details

This chapter provides the technical, syntax-level details of the NEDL and NSTL languages. I explain here every construct available in each of the languages and provide their use cases. Each language element is also followed by a corresponding piece of XML Schema to describe its structure and content formally. Finally, small examples of each language are given to help make their intended usage clear.

4.1 NEDL Syntax

The following sections provide an explanation of each of the elements allowed in a NEDL document, along with the relevant piece of XML Schema that defines element multiplicity and content type. Note that in the XML Schema snippets, the namespace prefix `tns` is used by convention to reference the same namespace specified by the `targetNamespace` attribute in the Schema. It is difficult to not make a few forward references to the validation process and parsing process described more fully in Chapters 6 and 5, respectively. For the time being, the reader should be aware that NEDL documents are validated against a schema written in the XML Schema language, and this schema is called the NEDL Schema. My project incorporates other validation procedures beyond just checking a NEDL document against the NEDL Schema. The module that performs these additional validation procedures will be temporarily referred to as "the NEDL validator." The module that performs the parsing of NEDL documents will be temporarily referred to as "the NEDL parser."

4.1.1 The `experimentspace` element

The `experimentspace` element must be the top-level element in a NEDL document. It must exist in the namespace identified by the following URI:

`http://www.eg.bucknell.edu/safe/exp`

The allowed sub-elements of the `experimentspace` root are `factorlist` and `conditions`. The `factorlist` element lists all of the factors that will be varied in the experiment, and the `conditions` element specifies what conditions the factor levels must meet to be considered a valid design point.

```
1 <xs:element name="experimentspace" type="tns:ExperimentSpace"/>
2 <xs:complexType name="ExperimentSpace">
3     <xs:sequence>
4         <xs:element ref="tns:factorlist"/>
5         <xs:element ref="tns:conditions"/>
6     </xs:sequence>
7 </xs:complexType>
```

4.1.2 The `factorlist` element

The `factorlist` element is the required first child of the `experimentspace` element. It specifies the set of experimental factors a researcher wishes to vary throughout the experiment. The element has no attributes. The only child element of `factorlist` is `factor`. Each `factor` element specifies the name of a factor (e.g., "ONTIME," "PACKETSIZE," "FREQUENCY," etc.)

```
1 <xs:element name="factorlist" type="tns:FactorList"/>
2 <xs:complexType name="FactorList">
3     <xs:sequence>
4         <xs:element ref="tns:factor" maxOccurs="unbounded"/>
5     </xs:sequence>
6 </xs:complexType>
```

4.1.3 The `conditions` element

The `conditions` element is the required second child of the `experimentspace` element. The `conditions` element has no attributes. The possible child elements of `conditions`, listed in the order in which they must appear are as follows:

- `memberof`, which specifies the name of a list of values that a given factor will take on in the experiment. This list can exist within the experiment description file itself or in an external file.
- `levellist`, an element that contains a simple list of level values which can be referenced by one or more `memberof` elements.
- `sequence`, which specifies a sequence of level values for a factor to take on in terms of a mathematical sequence.
- `exclusionrestriction`, which specifies certain "partial points" that the experimenter wishes to exclude in final design space.
- `linkingrestriction`, which specifies the factors which have a "one-to-one" level correspondence.

According to the NEDL Schema, each of these elements has a multiplicity of zero or more, though the NEDL validator checks that each factor listed in `factorlist` somehow has a specification of level values, through either the `memberof` or `sequence` elements.

```
1  <xs:element name="conditions" type="tns:Conditions"/>
2  <xs:complexType name="Conditions">
3      <xs:sequence>
4          <xs:element ref="tns:memberof"
5                      maxOccurs="unbounded" minOccurs="0"/>
6          <xs:element ref="tns:levellist"
7                      maxOccurs="unbounded" minOccurs="0"/>
8          <xs:element ref="tns:sequence"
9                      maxOccurs="unbounded" minOccurs="0"/>
10         <xs:element ref="tns:exclusionrestriction"
11                     maxOccurs="unbounded" minOccurs="0"/>
12         <xs:element ref="tns:linkingrestriction"
```

```
                    maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
```

4.1.4 The `memberof` element

The `memberof` element specifies a list of level values for a factor to take on during the experiment. The element has no attributes, but does have two required child elements, `factor` and `listid`. The `factor` element specifies the factor to associate with this list, and the `listid` element specifies *either* the pathname of an external XML file which conforms to the LevelList Schema or the `id` of a `levellist` element that specifies a simple list of level values (see Appendix A.3). Each have a multiplicity of 1. In the case that the content of `listid` matches *both* an internally-defined list and an external filename, the NEDL parser always uses the internally-defined `levellist`.

```
<xs:element name="memberof" type="tns:MemberOf"/>
<xs:complexType name="MemberOf">
    <xs:sequence>
        <xs:element ref="tns:factor"/>
        <xs:element ref="tns:listid"/>
    </xs:sequence>
</xs:complexType>
```

4.1.5 The `levellist` element

The `levellist` element provides a simple list of level values. It has one required attribute, `id`, which is an arbitrary identifier of the list, used to reference the list within a `memberof` element. The `levellist` element can have just one type of child element, the `level` element.

```
<xs:element name="levellist" type="tns:LevelList"/>
<xs:element name="level" type="xs:string"/>
<xs:complexType name="LevelList">
    <xs:sequence>
```

```
5           <xs:element ref="tns:level" maxOccurs="unbounded" minOccurs="0"/>
6       </xs:sequence>
7       <xs:attribute name="id" type="xs:string"/>
8   </xs:complexType>
```

4.1.6 The `sequence` element

The `sequence` element specifies a sequence of values based on some mathematical expression. The `sequence` element has no attributes. The child elements of `sequence`, in the order in which they must appear are:

- `factor`, which specifies the factor to associate with this sequence,
- `test`, an element whose value may either be `EQUALS`, `GT` or `LT` (which stand for "greater-than" and "less-than"),
- `lconst` *or* `lvar`, which specify either a constant numerical value or the variable being stepped in the sequence
- `op`, an element whose value may be one of the following mathematical operators:
  - "MULT" (multiplication)
  - "FDIV" (floating-point division)
  - "IDIV" (integer division)
  - "PLUS" (addition)
  - "MINUS" (subtraction)
  - "MOD" (modulo)
  - "POW" (power raising)
- `rconst` *or* `rvar`, which specify either a constant numerical value or the variable being stepped in the sequence
- `where`, an optional element which specifies the range of values for the stepping variable to take on. The required child element of `where` is the `range` element, whose required children are:
  - `var`, the variable to be stepped

- `lo`, the lower bound of the stepping variable
- `hi`, the upper bound of the stepping variable
- `delta`, the increment of the stepping variable

```
<xs:complexType name="Sequence">
    <xs:sequence>
        <xs:element ref="tns:factor"/>
        <xs:element ref="tns:test"/>
        <xs:choice>
            <xs:element ref="tns:lexpr"/>
            <xs:element ref="tns:lvar"/>
            <xs:element ref="tns:lconst"/>
        </xs:choice>
        <xs:element ref="tns:op"/>
        <xs:choice>
            <xs:element ref="tns:rexpr"/>
            <xs:element ref="tns:rvar"/>
            <xs:element ref="tns:rconst"/>
        </xs:choice>
        <xs:element ref="tns:where" maxOccurs="1" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
```

For factors whose levels are specified with *both* a `memberof` and `sequence` element, the resulting level list produced by the parser is the union of both.

4.1.7 The `exclusionrestriction` element

The `exclusionrestriction` element specifies a partial design point which should be considered invalid during design point generation. It has no attributes, and it's required child is the `setting` element. `setting` has two required attributes, `factor` and `level` which specify a particular factor/level combination considered invalid.

```
<xs:complexType name="ExclusionRestriction">
    <xs:sequence>
```

```
3           <xs:element ref="tns:setting" maxOccurs="unbounded"/>
4       </xs:sequence>
5   </xs:complexType>
6
7   <xs:element name="setting" type="tns:PointComponent"/>
8   <xs:complexType name="PointComponent">
9       <xs:attribute name="factor" type="xs:string"/>
10      <xs:attribute name="level" type="xs:string"/>
11  </xs:complexType>
```

4.1.8 The `linkingrestriction` element

The `linkingrestriction` element specifies a set of factors whose level values must correspond to the same index within their respective lists. It has one required child element, `factor`, which has a multiplicity of 1 or more.

```
1   <xs:complexType name="LinkingRestriction">
2       <xs:sequence>
3           <xs:element ref="tns:factor" maxOccurs="unbounded"/>
4       </xs:sequence>
5   </xs:complexType>
```

4.2 Details on the `sequence` element

The kind of sequence description the `sequence` element can provide is best illustrated with some examples. Suppose we have the factor `ONTIME` and would like to create for it a list of all multiples of 10 between 0 and 100. There are of course an infinite number of mathematical expressions that would generate this sequence, but one of them is

$$\text{ONTIME} = (x_0, x_1, x_2, \ldots, x_i, \ldots x_n)$$

where $x_i = 10 \times i$ and $n = 10$. More informally, and in a form more easily translated into the format of NEDL, we could say

$$\text{ONTIME} = 10 \times i \text{ where } 10 \leq i \leq 100$$

More complex sequences can be described as well, such as

$$\text{ONTIME} = \frac{i^2}{2} \text{ where } 1 \leq i \leq 4$$

which would give ONTIME = $(0.5, 2.0, 4.5, 8.0)$.

A snippet of XML used to describe the first sequence is as follows.

```
<sequence>
    <factor>ONTIME</factor>
    <test>EQUALS</test>
    <lvar>i</lvar>
    <op>MULT</op>
    <rconst>10</rconst>
    <where>
        <range>
            <var>i</var>
            <lo>0</lo>
            <hi>10</hi>
            <delta>1</delta>
        </range>
    </where>
</sequence>
```

A snippet of XML used to describe the second sequence is as follows.

```
<sequence>
    <factor>ONTIME</factor>
    <test>EQUALS</test>
    <lexpr>
        <lvar>i</lvar>
        <op>POW</op>
        <rconst>2</rconst>
    </lexpr>
    <op>FDIV</op>
    <rconst>2</rconst>
    <where>
        <range>
```

```
13              <var>i</var>
14              <lo>0</lo>
15              <hi>4</hi>
16              <delta>1</delta>
17          </range>
18      </where>
19  </sequence>
```

4.3 Details on the `exclusionrestriction` element

The `exclusionrestriction` element specifies individual design points an experimenter wishes to discard from the experiment design space. For example, suppose we have the following `exclusionrestriction` snippet.

```
1  <exclusionrestriction>
2      <setting factor="ONTIME" level="3.0"/>
3      <setting factor="OFFTIME" level="12.0"/>
4  </exclusionrestriction>
```

Then the design point

```
{ONTIME = 3, OFFTIME = 12, DELAY = 4}
```

would *not* be valid because it contains the `ONTIME = 3`, `OFFTIME = 12` combination, but the design point

```
{ONTIME = 2, OFFTIME = 12, DELAY = 4}
```

would be valid, because it does not contain an invalid combination.

4.4 Details on the `linkingrestriction` element

Every factor specified in a NEDL document has an associated list of level values. Each level value has a certain *index* within this list and the `linkingrestriction` element provides a way to specify a set of factors whose level values should all occur at the same index in their respective level lists. For example, suppose we have the following `linkingrestriction` element specified below.

```
1 <linkingrestriction>
2     <factor>ONTIME</factor>
3     <factor>DELAY</factor>
4 </linkingrestriction>
```

If the lists of level values associated with each factor are

ONTIME	DELAY
0	0
1	1.5
2	3
3	4.5

then the set of all possible valid design points would be

```
{ONTIME = 0, DELAY = 0.0}
{ONTIME = 1, DELAY = 1.5}
{ONTIME = 2, DELAY = 3.0}
{ONTIME = 3, DELAY = 4.5}
```

since the level values for `ONTIME` are "linked" with the level values for `DELAY`.

4.5 A Full Example

The following example describes an experiment space consisting of three factors, `ONTIME`, `OFFTIME` and `DELAY`. The level values for `ONTIME` are contained in the ex-

ternal file *onTimeValues.xml*, whereas the level values for `OFFTIME` are contained in an internally-defined `levellist` element. Finally, the level values for `DELAY` are expressed as $i \times 5$, where $0 \leq i \leq 10$.

There are two restrictions that prune the final set of design points. The first restriction is an `exclusionrestriction`, which specifies that any design point where

```
{ONTIME = 3, OFFTIME = 12}
```

is invalid. The second is a `linkingrestriction` between `ONTIME` and `DELAY`. With this restriction in place, valid design points are ones in which the level values of `ONTIME` and `DELAY` correspond to the same list index.

FullExample.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <experimentspace xmlns="http://www.eg.bucknell.edu/safe/exp">
3
4    <factorlist>
5        <factor>ONTIME</factor>
6        <factor>OFFTIME</factor>
7        <factor>DELAY</factor>
8    </factorlist>
9
10   <conditions>
11     <!-- Member-Of Constraints: -->
12
13     <!-- This associates ONTIME with an external list of level
14     values found in the ‘onTimeValues.xml’ file. -->
15     <memberof>
16         <factor>ONTIME</factor>
17         <listid>onTimeValues.xml</listid>
18     </memberof>
19
20     <!-- This associates OFFTIME with an internally-specified
21     list of level values found in the ‘levellist’ element whose
22     ‘id’ attribute is ’offTimeValues’ -->
```

```
23      <memberof>
24          <factor>OFFTIME</factor>
25          <listid>offTimeValues</listid>
26      </memberof>
27
28      <levellist id="offTimeValues">
29          <level>10</level>
30          <level>11</level>
31          <level>12</level>
32      </levellist>
33
34      <!-- Sequences -->
35      <!-- This associates DELAY with a sequence of values.
36           DELAY = i*1.5 where 0 <= i <= 10 -->
37      <sequence>
38          <factor>DELAY</factor>
39          <test>EQUALS</test>
40          <lvar>i</lvar>
41          <op>MULT</op>
42          <rconst>1.5</rconst>
43          <where>
44              <range>
45                  <var>i</var>
46                  <lo>0</lo>
47                  <hi>10</hi>
48                  <delta>1</delta>
49              </range>
50          </where>
51      </sequence>
52
53      <!-- Exclusion Restrictions -->
54      <!-- This type of restriction means that any design points
55      which have ONTIME = 3.0 and OFFTIME = 12.0 are invalid -->
56      <exclusionrestriction>
57          <setting factor="ONTIME" level="3.0"/>
58          <setting factor="OFFTIME" level="12.0"/>
59      </exclusionrestriction>
60
61      <!-- This type of restriction means that ONTIME and DELAY
```

```
62      should occur in one-to-one correspondence.  That is, a design
63      point is valid only if the level value of ONTIME is at the
64      same list index as the level value for DELAY -->
65      <linkingrestriction>
66          <factor>ONTIME</factor>
67          <factor>DELAY</factor>
68      </linkingrestriction>
69
70    </conditions>
71
72  </experimentspace>
```

onTimeValues.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <levellist xmlns="http://www.eg.bucknell.edu/safe/exp">
3      <level>0</level>
4      <level>1</level>
5      <level>2</level>
6      <level>3</level>
7  </levellist>
```

4.5.1 The Experiment Design Matrix

The experiment design matrix described by these documents is given in Table 4.1. Note that the `linkingrestriction` and `exclusionrestriction` constraints have reduced the number of valid design points drastically. Without these conditions there would have been a total of 132 design points (4 `ONTIME` points, 3 `OFFTIME` points and 11 `DELAY` points). With the restrictions in place, there are just 11.

4.6 NSTL Syntax

The following sections provide an explanation of each of the elements allowed in a NSTL document, along with the relevant piece of XML Schema that defines element

DELAY	OFFTIME	ONTIME
0.0	10.0	0.0
0.0	11.0	0.0
0.0	12.0	0.0
1.5	10.0	1.0
1.5	11.0	1.0
1.5	12.0	1.0
3.0	10.0	2.0
3.0	11.0	2.0
3.0	12.0	2.0
4.5	10.0	3.0
4.5	11.0	3.0

Table 4.1: A Full Design

multiplicity and content type. In keeping with my goal that NSTL be straightforward to use and understand, there are just three allowed elements in an NSTL document. Note that in the XML Schema snippets, the namespace prefix `tns` is used by convention to reference the same namespace specified by the `targetNamespace` attribute in the Schema.

4.6.1 The `template` element

The `template` element must be the top-level element in an NSTL document. It must exist in the namespace identified by the following URI:

`http://www.eg.bucknell.edu/safe/exp`

The only allowed sub-element of the `template` root is the `group` element. I envision that when researchers wish to create a new NSTL document they will first begin with an existing *ns-3* C++ script and annotate that script with NSTL tags after the fact. Thus the content type of a `template` element is "mixed," meaning that it can contain both text and other elements. The allows for `group` elements to be inserted in and around the code of an *ns-3* script.

```
<xs:element name="template" type="tns:Template"/>
<xs:complexType name="Template" mixed="true">
    <xs:sequence>
        <xs:element ref="tns:group" minOccurs="1"
                                    maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

4.6.2 The `group` element

The `group` element is the required child of the `template` element and there may be an unlimited number of occurrences of the element. A `group` element has one required child element: the `block element`. The idea is that the `group` element will contain a number of swappable code blocks, each within their own `block` element.

```
<xs:element name="group" type="tns:Group"/>
<xs:complexType name="Group">
    <xs:sequence>
        <xs:element ref="tns:block" minOccurs="1"
                                    maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id"/>
</xs:complexType>
```

4.6.3 The `block` element

The `block` element is intended to contain some block of *ns-3* script code which the user wishes to swap out with other code blocks over the course of the experiment. As stated previously, each of these `block` elements should be within a single `group` for proper code generation.

```
<xs:element name="block" type="xs:string"/>
```

A full example to better illustrate the use of NSTL is in order, but this is given later in Chapter 7 on code generation. For now, the important points worth repeating

are that NSTL has been pared back as a model description language and now serves a purpose better described as template description language. It allows one to annotate an existing *ns-3* script with special NSTL markers that facilitate the swapping of certain code blocks within a section of code (these sections are delimited by `group` elements). If a researcher wished to use two different Application Layer models in their simulation script, they could instantiate each of those models in a different `block` element and run simulations comparing the two.

Chapter Summary

This chapter included the syntax-level details of both languages I have developed, NEDL and NSTL. NEDL is somewhat more complex and allows for an abstract description of an experiment design space. The language also has special constructs (i.e., `exclusionrestriction` and `linkingrestriction`) that provide ways to prune unwanted design points from the experiment design space.

NSTL has a simple grammar; it augments the code of an existing *ns-3* simulation script with annotations to facilitate text replacement. Its structures allow one to delimit sections of code with `group` elements and swap blocks of code within that section by placing the blocks within `block` elements. Although NSTL does not allow for "simulator-agnostic" high-level model descriptions, it provides a flexible and useful templating mechanism for users of the *ns-3* simulator. Chapter 7 discusses the challenges identified in my experience in generating *ns-3* code from high-level model descriptions.

Chapter 5

Parsing NEDL and NSTL Documents

This chapter discusses the larger points related to manipulating documents written in XML-based languages, such as NEDL and NSTL. The only way to access information programmatically within an XML file is to first parse it, and the act is so common that there exist a variety of XML parsing libaries in every major programming language. These libraries present an interface to the programmer, and it is through this interface that the document can be read and written to. There are three main approaches for parsing an XML file. The first involves parsing the entire document and then building a tree data structure in memory that matches the tree structure of the document. In this approach, the programmer is presented with the Document Object Model (DOM) interface. The second approach involves parsing the document and firing events as certain elements are encountered. In this approach, the programmer is presented with the Simple API for XML (SAX) interface. In the last approach, the document is parsed and used to instantiate a custom-designed object in memory that matches the element names and attributes of the document. This last approach is called data binding, and is perhaps the most programmer-friendly of the three [28] [45].

5.1 Document Object Model

To create the DOM interface, the entire XML document is read into memory and placed in a tree data structure. The structure of XML documents corresponds naturally to a tree hierarchy with parent and child node relationships. The programmer is provided a reference to the top-level element in the document (the *root*) and may reference other positions in the document (called *nodes*) using either special method calls like `getFirstChild()` (which returns a reference to the first child element appearing underneath the root). If the programmer has a reference to some node underneath the root, they could call methods like `getNextSibling()`, which returns a reference to the next node in the document whose parent is the same as the current node's.

The DOM interface is usually easier for those less familiar with XML processing as it allows for random access to any part of the XML document tree using intuitive methods and names. The DOM parser that constructs the tree in main memory, however, requires substantial resources in terms of CPU time and memory. Memory requirements are often 2 to 5 times the size of the document itself, which limits the scalability of DOM parsing as the XML document size grows [36].

5.2 Simple API for XML

The Simple API for XML (SAX) provides an event-based interface to the programmer. SAX parsers read through an XML document and report events to the programmer through callbacks. For example, a `startElement` event will fire when the parser hits the opening tag of an element, and an `endElement` event will fire when the parser reaches the closing tag. The programmer constructs event handlers in their application to process information as it is read from the document. Since a SAX parser delivers an event stream to the programmer, memory requirements remain constant even as the XML document size increases [36]. Additionally, SAX parsers can process a large document between 2 and 7 times faster than a DOM parser [21]. The drawback to SAX parsers is their complexity; handling an event stream is conceptually much different than working with a document tree. For this reason, SAX parsers can be difficult to build and maintain.

5.3 Data Binding

Data binding, a third approach for parsing XML documents, is the approach I chose in implementing the relevant pieces of the SAFE framework. Both the SAX and DOM interface to XML documents are very low level and document-centered [28] [45]. It can be tedious work, since the transformation of data to and from XML has to be done manually. A more streamlined approach is to allow the programmer to work with objects that better reflect the structure of the application domain rather than that of the underlying tree structure of XML [45]. This is accomplished by creating custom software objects that correspond to the thing an XML document is intended to represent and creating data members for each of the document's XML elements. Programmers used to an object-oriented paradigm find it natural to interact with XML documents in this fashion. This not only makes things easier for the programmer, but also makes sense from a software engineering standpoint; software objects provide the semantics that XML by itself does not.

Every major programming language has XML data binding libraries available which are able to transform a given XML document into an object-oriented data structure. To do this, the libraries must first know what structure and content to expect in the document, and this information is provided in an XML schema given as input to the data binding program. The output of that program is a set of object classes corresponding to the schema [44]. The schema language that has the widest support is the W3C XML Schema language. Every XML Schema defines a certain class of XML documents, and generating the programming code to represent that class of documents is straight-forward and easily automated [44].

Once the object classes have been generated we can *unmarshal* XML documents into instances of these classes. Since the SAFE framework is implemented in Python and C++, it made sense to unmarshal our XML documents into instances of Python classes. The *PyXB* library [1] was my library of choice. From the command line we can simply run

```
$ pyxbgen -u MySchema.xsd -m mymodule.py
```

and PyXB will automatically generate the `mymodule.py` file, which defines a Python class corresponding to the structure of `MySchema.xsd`. If we have the XML file `MyDocument.xml` which validates against `MySchema.xsd`, then a call to

```
mymodule.CreateFromDocument(file('MyDocument.xml').read())
```

instantiates a Python object whose instance variables correspond exactly to the structure and content of the XML document's elements and attributes.

The great benefit of this is that we now have object-oriented access to the XML document contents and are saved from having to write custom SAX or DOM parsers by hand. As an aside, the `CreateFromDocument()` method generated by PyXB in fact uses a custom-built SAX parser to incrementally construct a class instance from an XML document [1]. If the definition in the original schema changes (e.g., a change in `MySchema.xsd`) then we need only to re-run the `pyxbgen` tool to generate a new binding module. To do this by hand would mean re-factoring the original SAX parser every time the schema definition evolved, which is tedious and error-prone work.

5.4 Advantages to Data Binding

In the context of the SAFE framework for *ns-3*, the XML documents that we are parsing and binding to Python objects are NEDL files for experiment description and NSTL files for template description. For the moment, I will concentrate on the processing associated with a given NEDL file.

As mentioned in Chapter 1, an experiment design space is composed of a set of design points. The goal in my work is to take a NEDL file as input and generate each design point within the set of all design points described by that document. Data binding provides a way to modularize this process. In writing a DOM or SAX parser by hand it would be tempting to generate these design points *as* the document is parsed, and in fact, it may even be more efficient that way. The problem with this approach is that if either the inputs or outputs to the process change, the entire parsing module would have be re-factored and re-compiled by hand. As shown in Figure 5.1, the binding classes generated by PyXB allow for a separation of the parsing and experiment space generation logic.

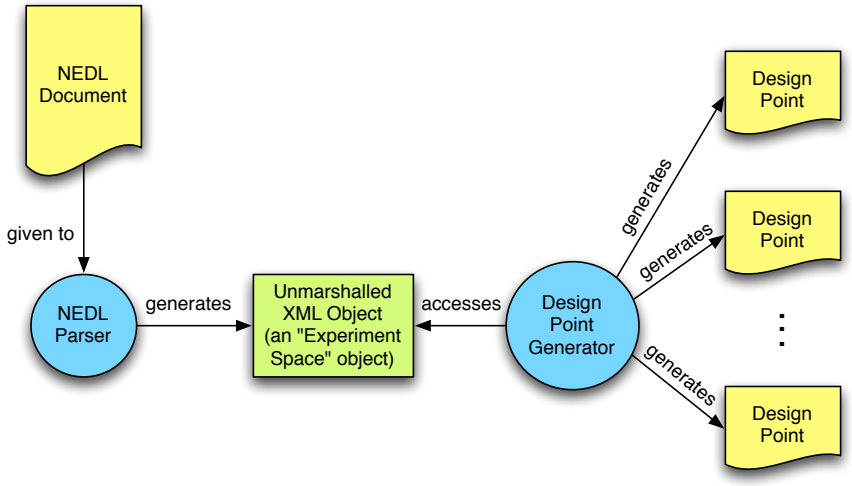

Figure 5.1: From NEDL document to design point generation.

Of course, for now, the names of instance variables that are accessed by the **NEDL Parser** module depend on element and attribute names defined in the NEDL Schema, so if the input Schema were to change, the parsing module would have to be re-factored as well. In most data binding libraries, those could be modified with a special bindings file that specifies names of XML elements and what their corresponding instance variable names should be in the generated class. This functionality is not supported by PyXB, but implementing it would not be unreasonably difficult, it is safe to assume that if this functionality is not supported in future versions of PyXB, another binding library will support this. If one were to tweak the NEDL Schema in a way that would only affect element or attribute names, then they could make similar changes in the bindings file and re-run the `pyxbgen` tool without having to make any changes in the **NEDL Parser** module. On the other end, if one wished to change the internal logic of the **NEDL Parser** module or change any of its method signatures, they could do so without having to interfere with any code relating to syntax-level document parsing. These advantages also apply to the parsing of NSTL documents, though the issue of an evolving structure is less urgent in that case since NSTL documents have a much simpler structure than NEDL documents.

5.5 Alternative Parsing Methods

There are other "hybrid" parsing methods available, notably the RELAX NG Compiler-Compiler (RELAX NGCC) technology [38]. This technology allows a user to annotate a schema written in the RELAX NG language with lines of Java source code that will execute *as* the document is being parsed and validated against the schema. It does this by reading the annotated RELAX NG schema and generating a dedicated SAX parser that integrates the Java code embedded in the schema into the call-back methods. For example, the RELAX NGCC website provides the following small grammar:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <grammar xmlns="http://relaxng.org/ns/structure/1.0"
3    datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
4    xmlns:c="http://www.xml.gr.jp/xmlns/relaxngcc">
5
6  <start c:class="Driver">
7
8  <element name="team">
9    <oneOrMore>
10     <element name="player">
11       <attribute name="number">
12         <data type="positiveInteger" c:alias="number"/>
13         <c:java>System.out.println(number);</c:java>
14       </attribute>
15       <element name="name">
16         <text c:alias="name"/>
17         <c:java>System.out.println(name);</c:java>
18       </element>
19     </element>
20   </oneOrMore>
21 </element>
22
23 </start>
24
25 </grammar>
```

Listing 5.1: An example grammar using the RELAX NGCC syntax.

The grammar describes a team roster written in XML that would look something like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- this file is "Team.xml" -->
<team>
    <player number="23">
        <name>John Smith</name>
    </player>
    <player number="11">
        <name>Bob Jones</name>
    </player>
</team>
```

Listing 5.2: An example instance document for use with RELAX NGCC.

The tools provided by the RELAX NGCC project allow one to compile the grammar given in Listing 5.5 into the source code for a SAX parser that makes the calls to `System.out.println()` whenever the `number` attribute or `name` element are reached in the instance document. This source code can then be compiled with the standard Java compiler into executable Java byte code whose input is the XML instance document and whose output (in this case) would be:

```
$ java Driver Team.xml
23
John Smith
11
Bob Jones
```

This compiler-compiler approach is helpful because it is able to associate behavior with XML documents through the use of annotations in the schema document [28]. My work does not utilize the RELAX NGCC technology largely because it is intended for generating Java source code from a given grammar [38] and I believed that the *ns-3* community would rather see the implementation of the SAFE framework done in a language they already use and are familiar with (i.e., Python or C++). Like the RELAX NGCC approach, my work uses data binding libraries to build custom parsers too. The difference however, is that RELAX NGCC allows a programmer to specify special actions for the parser to take as it parses the document, whereas in

my approach, the document is first parsed and used to populate an object instance in memory. From there, special actions can be taken on the values of its instance variables. The compiler-compiler approach is perhaps more efficient, but unmarshalling with PyXB is suitable too, since it provides the important benefit of separation of low-level syntax processing code and application logic.

Chapter Summary

This chapter discusses XML parsing techniques as they relate to SAFE. Parsing is a necessary step for interacting with an XML document in any programmatic way. There are three main ways to go about this, and they involved working with different interfaces to the XML document. Using the DOM interface, the programmer is presented with the entire document tree in memory, and has random access to any node in the tree. Using the SAX interface, the programmer is presented with an event stream and is responsible for writing handler methods to operate on the document contents appropriately. Although the DOM interface is easier for most beginners to use, it consumes an amount of memory proportional to the document and therefore does not scale well with large XML documents.

A third approach is data-binding, which is the process of using an XML schema to generate a class file that encapsulate the information which the XML document is intended to represent. Given this class file, a data-binding program can *unmarshal* the document into an instance of that class. This instantiated object is much easier for a programmer to use because it uses represents the document as an abstract data type rather than tree of nodes or stream of events. I have used the PyXB data-binding library because I believe it is a reasonably-efficient parsing method and makes the experiment space generation and template generation logic much easier to understand.

Chapter 6

Validating NEDL and NSTL Documents

A crucial step in the SAFE framework is validation of NEDL and NSTL files. Developing XML-based languages was a decision motivated largely by the existence of so many tools and methods available for document validation, and as such, the topic of validation merits further discussion here. In this chapter, I talk about some of the ways an XML document may be validated, including grammar-based and rules-based methods. There are many languages available to describe XML grammars and it is advantageous to develop a grammar for the purposes of data-binding, as described in Chapter 5. It is difficult, however, to capture all of the semantic rules of a language in a grammar, and for this reason I have also implemented a tool that performs something like rules-based validation on top of a grammar-based validation.

6.1 Introduction

An *XML schema* is a rigorous specification of the allowed elements and attributes in a particular XML-based language. The schema defines constraints on what elements and attributes are allowed in the language, and how they should be structured [33]. We say that an XML document is valid against a schema if it does not break any of the constraints laid out in that schema. The computer languages used to encode a schema are known as *schema languages*. A *validator* for a schema language is a

computer program that takes as input both an XML schema written in that language and an XML document, and subsequently determines if the document is valid against the schema.

There are several schema languages at our disposal, including the general-purpose Document Type Definition (DTD) [11], W3C XML Schema [47], and RELAX NG [16]. Although other languages do exist for specialized applications or research activities (e.g., XDuce, DSD, RDF Schema), these are not well-suited for this project. Moreover, as described in my review of data-binding in Chapter 5, there is an advantage to pursuing the more widely-supported schema languages. Worth noting, however, is ISO Schematron [2], which uses a system of *rules-based* validation to check the validity of a document. In rules-based validation, the XML instance document is simply subjected to a series of "checks" (these are the rules) to decide if it is valid or not. The rules can be very specific, allowing for tight control over the document structure and contents. One could, for example, specify that the contents of an XML element of the form `<bogus n="2">4</bogus>` always has numerical content value that is double the value of the `n` attribute. Although I do not integrate ISO Schematron itself into my work, I have borrowed concepts from its rules-based scheme in my own validation process.

Murata et al. [33] have completed substantial work in the area of XML validation using formal language theory. Their research has influenced the design choices I have made regarding the specific methods for validation and the schema language used for defining valid NEDL and NSTL documents.

6.2 XML and Tree Grammars

Of the two languages I have developed, NEDL is the most difficult to validate. This is because there are many constructs within a NEDL document for which validation is hard. Consider, for example, the NEDL excerpt shown below.

```
<factorlist>
    <factor>DELAY</factor>
    <factor>FREQ</factor>
</factorlist>
<conditions>
    <memberof>
        <factor>ONTIME</factor>
        <listid>bogus_id</listid>
    </memberof>
</conditions>
```

The NEDL Schema specifies that a `memberof` element must have the `factor` and `listid` elements as children, but there is no way to specify in the schema that the value of `factor` is a string that appears in the original `factorlist` and that `listid` marches either a `levellist` element's `id` attribute or an external filename. The problem here is XML Schema's limited expressivity as a language for describing grammars.

Murata et al. [33] give rigor to our discussion by introducing *tree grammars*, which are grammars that generate trees. These should not be confused with the perhaps more familiar context-free grammars, which generate strings. Since an XML document is a representation of a tree structure, tree grammars are more appropriate when talking about XML. The class of tree grammars that are of interest to us are *regular* tree grammars. This class can be further divided into *local* and *single-type* grammars.

First, a formal definition of regular tree grammars. This definition is borrowed from Comon et al. [17]. A regular tree grammar is a 4-tuple $G = (N, T, S, P)$ where

- N is a finite set of nonterminals,
- T is a finite set of terminals,
- S is a set of start symbols, where S is a subset of N,
- P is the set of production rules of the form $X \rightarrow \mathbf{a}r$ where $X \in N$, $\mathbf{a} \in T$ and r is a regular expression over N; X is the left-hand side, $\mathbf{a}r$ is the right-hand side, and r is the called the *content model* of this production rule.

Using this framework Murata et al. [33] are able to classify the relative power of the three important schema languages I listed earlier: DTD, XML Schema and RELAX NG. The power they talk about comes from each of the language's expressivity and to what degree they allow for *competition* among nonterminals and terminals in a grammar G. Competition among two different nonterminals occurs when two production rules that have different left-hand sides share the same terminal in their right-hand sides. Given the following production rules (where * is the Kleene-star operator)

$$\begin{aligned} \text{BOOK} &\rightarrow \textbf{author}\ \text{CHAPTER}^* \\ \text{REPORT} &\rightarrow \textbf{author}\ \text{ABSTRACT} \end{aligned}$$

the nonterminals BOOK and REPORT are in competition with each other. Validating an XML document against this grammar is more difficult, since the validator must somehow decide which production rule to use when comparing an XML tree against this grammar.

DTD is the least expressive schema language because it prohibits competition of nonterminals and is only able to describe *local tree grammars*. XML Schema expands on the expressivity of DTD by allowing for competition among nonterminals, as long as they do not exist within the same single content model. This class of grammars is called *single type*. For example, the production rules

$$\begin{aligned} \text{COLLECTION} &\rightarrow \text{BOOK REPORT} \\ \text{BOOK} &\rightarrow \textbf{author}\ \text{CHAPTER}^* \\ \text{REPORT} &\rightarrow \textbf{author}\ \text{ABSTRACT} \end{aligned}$$

could not be part of a single-type grammar, because both BOOK and REPORT are part of the content model of COLLECTION, and BOOK and REPORT are in competition with each other. However, if the rules were modified so that we had

$$\begin{aligned} \text{COLLECTION} &\rightarrow \text{B R} \\ \text{B} &\rightarrow \textbf{booktitle}\ \text{BOOK} \\ \text{R} &\rightarrow \textbf{reporttitle}\ \text{REPORT} \\ \text{BOOK} &\rightarrow \textbf{author}\ \text{CHAPTER}^* \\ \text{REPORT} &\rightarrow \textbf{author}\ \text{ABSTRACT} \end{aligned}$$

then the grammar would be single-type. Nonterminals BOOK and REPORT are indeed in competition with each other, but they are not part of the same content model. The class of single-type grammar corresponds roughly to XML Schema.

RELAX NG broadens this expressivity further still by allowing for competition among nonterminals in any context. For this reason, it is generally considered to be the most powerful schema language of those I am considering here. RELAX NG is able to encode the production rules

$$\begin{aligned} \text{BOOK} &\rightarrow \textbf{author}\ \text{CHAPTER}^* \\ \text{REPORT} &\rightarrow \textbf{author}\ \text{ABSTRACT} \end{aligned}$$

and validators exist to validate a document conforming to this grammar. Despite RELAX NG's expressive power, it does not enjoy the same widespread use that XML Schema has across many types of applications, due in part to XML Schema's official endorsement by the World Web Consortium. By incorporating elements of rules-based validation into the overall validation process we can mitigate the limited expressivity of XML Schema while leveraging the wide support is has as the de-facto schema language standard in the XML community.

To this end, a NEDL document is first validated against the NEDL Schema written in XML Schema. XML well-formedness checking is implicit in this process. As shown earlier however, there are a number of variations that are valid according to the NEDL Schema but meaningless or nonsensical from the perspective of the NEDL parser. I have written the `NEDLValidator` class to perform the sort of rules-based validation alluded to earlier that can check for the presence of specific content in the document and ensure that it is semantically correct.

6.3 Implementation

The implementation of the `NEDLValidator` class is relatively straight-forward. The hierarchical structure of XML makes it easy to perform element-by-element validation. The `NEDLValidator` class is initialized with an XML document string (that is, a string containing an entire XML document). The constructor uses a PyXB-generated module to unmarshal the document into a Python class. The method `ExpValidator.validate()` begins the validation process by accessing XML elements and contents in their unmarshalled form.

Recall that a NEDL document has the following form outlined below.

```
1  <experimentspace xmlns="http://www.eg.bucknell.edu/safe/exp">
2
3    <factorlist>
4        <factor>DATARATE</factor>
5        <factor>DELAY</factor>
6        <factor>ONTIMEMEAN</factor>
7        <factor>ONTIMEVARIANCE</factor>
8    </factorlist>
9
10   <conditions>
11       <!-- ... -->
12   </conditions>
13
14 </experimentspace>
```

The `NEDLValidator.validate()` method first iterates through the `factor` elements within `factorlist` and constructs a list data structure to hold each of the factor names. Subsequently, a series of methods are called that correspond exactly to the XML document structure. Since every NEDL document has a `conditions` element immediately after `factorlist`, the `NEDLValidator.validate_conditions()` method is called next, which itself simply calls `NEDLValidator.validate_memberof()` and `NEDLValidator.validate_sequence()`. The `NEDLValidator` class has a method name of the form `validate_<ELEMENT NAME>` for every complex type element in a NEDL document. The next section describes what kind of checking is performed in the validation of each element in a NEDL document. Because NSTL documents are essentially just augmented *ns-3* scripts, attempting to validate an NSTL document at a granularity any finer than basic element structure would be tantamount to validating the C++ code itself – a task better suited to existing compilers. Thus, NSTL documents are validated only against the NSTL Schema (see Appendix A.2) and there is no further semantic validation performed.

6.4 NEDL Checking

The `NEDLValidator` class contains methods for checking each type of element that may be in a NEDL document. There are four methods used to check the most important elements in the document. These methods are used for

- validating `memberof` elements,
- validating `sequence` elements,
- validating `linkingrestriction` elements, and
- validating `exclusionrestriction` elements.

6.4.1 Checking `memberof` elements

The `memberof` element has children `factor` and `listid`. A typical `memberof` element has the following form:

```
1 <memberof>
2     <factor>ONTIME</factor>
3     <listid>onTimeValues</listid>
4 <memberof>
```

The `NEDLValidator` checks that the `factor` element matches one of those listed in the `factorlist` element. The validator also checks that `listid` matches *either* (1) the `id` attribute of some `levellist` element or (2) the name of a file in the users current working directory.

6.4.2 Checking `sequence` elements

NEDL `sequence` elements are more complex to validate than `memberof` elements because there is greater flexibility in the NEDL Schema to allow for grammatically-correct but semantically-incorrect element contents. Recall that a typical `sequence` element has the following form:

```
<sequence>
    <factor>ONTIME</factor>
    <test>EQUALS</test>
    <lconst>2</lconst>
    <op>PLUS</op>
    <rvar>i</rvar>
    <where>
        <range>
            <var>i</var>
            <lo>100</lo>
            <hi>110</hi>
            <delta>1</delta>
        </range>
    </where>
</sequence>
```

The general validation process executed by the `NEDLValidator` for these elements is detailed below. Note that from the point of view of the `NEDLValidator`, the `lexpr` and `rexpr` represent "expression" elements, which can themselves contain further nested expressions. Loosely speaking, these expression elements contain a left-hand side, an operator, and a right-hand side. See the NEDL Schema for details on this feature.

1. Verify that the `factor` element matches one of those listed in the original `factorlist` element.
2. Verify that the value of the `test` element is one of the three strings "EQUALS", "LT" or "GT".
3. Validate the next `lconst`, `lvar` or `lexpr` element, depending on which is present.
    - `lconst`: Check that the value of this element is an integer or float.
    - `lvar`: If this is the first `lvar` or `rvar` element encountered, `NEDLValidator` saves the value of this string. For the sake of simplicity, only single-variable expressions are allowed in NEDL files, so all subsequent `lvar` and `rvar` elements are compared to this first string. If the value differs from the initial variable identifier, `NEDLValidator` issues a warning, but continues on as if the variable identifiers *were* all the same.

- `lexpr`: Recursively validate this as a nested "expression" element.

4. Verify that the value of the `op` element is one of the strings "MULT", "FDIV", "IDIV", "PLUS", "MINUS", "MOD", or "POW".
5. Validate the next `rconst`, `rvar` or `rexpr` element, depending on which is present. Following the exact same procedure described in Step 3.
6. Validate the `where` element by verifying that the nested `lo`, `hi`, and `delta` elements contain valid integers or floating-point numbers.

Subsequently, `NEDLValidator` checks that the value of the `op` element is one of the strings Finally, the checks performed for the `rconst`, `rvar` and `rexpr` elements are exactly the same as described above for the `lconst`, `lvar` and `lexpr` elements.

6.4.3 Checking `linkingrestriction` elements

A typical `linkingrestriction` element has the following form:

```
<linkingrestriction>
    <factor>ONTIME</factor>
    <factor>OFFTIME</factor>
    <factor>DELAY</factor>
    <factor>DATARATE</factor>
</linkingrestriction>
```

The `NEDLValidator` simply checks that each `factor` element matches one of those listed in the original `factorlist` element.

6.4.4 Checking `exclusionrestriction` elements

A typical `exclusionrestriction` element has the following form:

```
<exclusionrestriction>
    <setting factor="ONTIME" level="1.0"/>
```

```
3       <setting factor="OFFTIME" level="3.0"/>
4   </exclusionrestriction>
```

The `NEDLValidator` checks that the value of each `factor` attribute matches the value of one of `factor` elements in the original `factorlist` element. The validator also verifies that value of the `setting` attribute is a float or integer.

6.5 Chapter Summary

Validation is a crucial piece of the SAFE framework and the various validation options explored over the course of this project were discussed in this chapter. We have a range of schema languages to choose from when deciding how to encode the grammar of an XML language. Among them, the most common are DTD, XML Schema and RELAX NG, which are each able to describe increasingly larger subsets of the class of *regular tree grammars*. The DTD language corresponds roughly to the class of *local tree grammars*, the XML Schema language corresponds roughly to the class of *single-type grammars*, and RELAX NG is able to describe the entire class of *regular tree grammars*. Although RELAX NG represents the greatest degree of expressivity over an XML grammar, it does not have the widespread adoption that XML Schema has enjoyed. Because XML Schema is so well integrated into all sorts of XML applications, there exist a variety of robust libraries and tools that depend on the language. Data-binding, in particular, is often carried out using an XML Schema. To account for the instances in which XML Schema is insufficient for describing a valid NEDL document, I have written the `NEDLValidator` tool which carries out a series of checks to ensure that a given NEDL document is semantically correct and ready to be parsed.

Chapter 7

Code Generation

Automatic code generation is one of the higher-level goals of the SAFE project and has been perhaps the most ambitious goal of the project. I have concentrated on providing a language that facilitates using code *templates* in flexible ways. Although the pool of researchers who know the codebase of *ns-3* well is small, if these people help develop several dozen script templates then the larger group of all network simulation researchers will have a collection of robust scripts that allow for interchangeable code blocks and changes in model input parameters. This chapter gives some further background to code generation and how it is typically performed in XML applications, along with a description of the specific steps I took in my own code generation scheme.

7.1 Introduction

The NEDL language provides ways to describe design spaces completely and relatively succinctly. When the experiment description document has been validated and parsed, it is ready to be given to the design point generator module. Two classes written in Python, `safe.dpgenerators.LinearDesignPointGenerator` and `safe.dpgenerators.BacktrackingDesignPointGenerator` perform this design point generation. In the SAFE framework, a single design point is represented as a Python dictionary, where the keys of the dictionary are factor names and the values are the corresponding level values for these factors. Given a single design point, the next step in the process is code generation. Code generation is generally the process of

transforming a high-level document into a low-level one [24].

The high-level language in this case is the XML language NEDL and the lower-level target language is *ns-3*-executable C++. Though the input language is XML this process should not be confused with *data binding*. We do not wish to instantiate objects in the *ns-3* library — we are generating flat text files containing C++ scripts that each correspond to a single simulation that is part of a many-simulation experiment.

7.2 Working with Design Points

The simplest sort of code generation scheme is one involving string substitution. For example, suppose we have a snippet of a simulation script that is the following.

```
1 OnOffHelper onoff ("ns3::UdpSocketFactory",
2     Address (InetSocketAddress (Ipv4Address ("10.1.1.2"), port)));
3 onoff.SetAttribute (
4     "OnTime", RandomVariableValue (ConstantVariable (1)));
5 onoff.SetAttribute (
6     "OffTime", RandomVariableValue (ConstantVariable (0)));
```

If a researcher wanted to vary the values used for the `OnTime` and `OffTime` attributes, they could use the NEDL language described previously to generate a series of design points with different level values for `OnTime` and `OffTime`. One way of transforming these design points into executable code is to examine an existent simulation script and find all the places where each factor in the design space is set. Using the level value contained in the design point, it is only a matter of substituting this value into the existent script and outputting the result to a user-specified file.

In this project I chose to use special markers within a simulation script to identify places where such substitution would occur. These markers are identified by a pair of dollar sign (`$`) symbols. Thus, if we had a NEDL document that set up a design space with factors `ONTIME` and `OFFTIME`, then the previous piece of code setting up the `OnOffHelper` object would look like:

```
1 OnOffHelper onoff ("ns3::UdpSocketFactory",
2     Address (InetSocketAddress (Ipv4Address ("10.1.1.2"), port)));
3 onoff.SetAttribute (
4     "OnTime", RandomVariableValue (ConstantVariable ($ONTIME$)));
5 onoff.SetAttribute (
6     "OffTime", RandomVariableValue (ConstantVariable ($OFFTIME$)));
```

The code generating module in this case has only to pluck the level values corresponding to the `ONTIME` and `OFFTIME` factors (recall that this information is contained in a Python dictionary) and substitute these values in place of the markers. This functionality, while quite trivial to implement, already provides a great deal of flexibility to the experimenter. No extra special command-line arguments are needed to run each of the outputted scripts — the markers have already been replaced by hard-coded values. The *ns-3* simulator is billed as a simulator used for both education and research [37], and one use case that this functionality targets is that of a professor providing students with a pre-built *ns-3* script with these special factor markers embedded inside. The students could easily design their own experiments using the NEDL language and observe how simulator behavior changes after running each script.

7.3 XML Transformations

Any code generation that is more complex than simply value substitution involves the transformation of XML documents into some other form. It is "compiling" XML into the target language. XML-to-C++ transformations could be done in the following ways:

- by hand, using a SAX or DOM parser and custom-designed handlers that print out the right C++ code to an output file;
- by hand, using a specification written in the eXtensible Stylesheets Transformation Language (XSLT).

7.3.1 Using XSLT

XSLT is an oft-used language for XML document transformation; its specification is an official recommendation of the World Wide Web Consortium [15] (as is XML [11]). Processing in XSLT is divided into special *templates*. Each template can either be called explicitly or invoked whenever a certain XPath expression matches a node in the input document. It is best suited, however, for transforming one XML document into another XML document, as opposed to transforming that same XML document into an arbitrary text file (such as C++ source code).

Projects that have used XSLT for source code generation typically develop templates that consist mostly of blocks of the target code with small pieces of XSLT that "plug in" values from the input XML document [48] [22] [13]. For example, consider the XSLT template in Listing 7.1 offered as an example in [22] where the XSLT processing information is highlighted in gray, and the surrounding text that is simply dumped as output is in black. Another example from [48] is given in Listing 7.2. Finally, a third example taken from Canonico et al. [13], whose work lies squarely in the domain of network simulation, is given in Listing 7.3.

These examples illustrate some of the major drawbacks of XSLT. For one, it is itself written in XML and is rather verbose. Also, it is not always immediately obvious what exactly the XSLT code is accomplishing. The programmer is forced to write what are sometimes cumbersome XPath expressions and the for-loop detailed in Listing 7.2 is not formatted in a way most programmers are used to seeing (i.e., no special indentation for the loop body or braces). These examples are straight-forward enough, but a more complex transformation can become nearly indecipherable, and if the specification of the input XML document changes, maintenance of the stylesheet is all the more difficult. It seems that most kinds of code generation using XSLT use the same sort of concept as [48] and [22]: manually create the skeleton code block of the target language and use `<xsl:value-of/>` elements to pluck element contents from the input XML document.

```
<xsl:template match="RemoteObject" mode="connect">
try {
    ORB orb = org.omg.CORBA.ORB.init(args, null);
    org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");
    NamingContext ncRef = NamingContextHelper.narrow(objRef);
    NameComponent nc =
        new NameComponent("<xsl:value-of select="."/>", " ");
    NameComponent path[] = {nc};
    <xsl:value-of select="."/>_SINGLETON =
    <xsl:value-of select="../RemoteServer"/>Lib.
    <xsl:value-of select="."/> Helper.narrow(
    ncRef.resolve(path));
} catch(Exception e) {
    System.out.println("ERROR : " + e);
    e.printStackTrace(System.out);
}
</xsl:template>
```

Listing 7.1: An XSLT template from Grundy et al. [22] used for generating source code.

```
// shutdown all our connections
int infopipe_<xsl:value-of select="$thisPipeName"/>_shutdown()
{
<jpt:pipe point="shutdown">
// shutdown incoming ports <xsl:for-each select="./ports/inport">
infopipe_<xsl:value-of select="@name"/>_shutdown(); </xsl:for-each>
// shutdown outgoing ports <xsl:for-each select="./ports/outport">
infopipe_<xsl:value-of select="@name"/>}_shutdown(); </xsl:for-each>
</jpt:pipe>
return 0;
}
```

Listing 7.2: An XSLT template from Swint et al. [48] used for generating source code.

```
<xsl:template match="/">
#Code Generated by SimulationScenarioTons.xsl
set ns [new Simulator]
<xsl:apply-templates
     select="//sce:simulationCommand/sce:
     nsConfigurationCommand/sce:traceall"/>
<xsl:apply-templates
     select="//sce:networkDescription"/>
<xsl:apply-templates
     select="//sce:traffic"/>
#finish proc
proc finish {} {
global ns
$ns flush-trace
<xsl:if
     test="//sce:simulationCommand/sce:
     nsConfigurationCommand/sce:traceall/@nam='false'">
global tf
close $tf
</xsl:if>
<xsl:if
     test="//sce:simulationCommand/sce:
     nsConfigurationCommand/sce:traceall/@nam='true'">
global nf
close $nf
</xsl:if>
exit 0
}
#Simulation Run
$ns at <xsl:value-of select="//sce:simulationCommand/@stopTime"/>
"finish"
$ns run
</xsl:template>
</xsl:stylesheet>
```

Listing 7.3: An XSLT template from Canonico et al. [13] used for generating source code for the *ns-2* network simulator.

7.3.2 An Alternative Approach: Groups and Blocks

Having spent a good deal of time working with XSLT, I believe there is a better approach for generating source code from XML. As mentioned previously, the key observation to be made is that most kinds of source code generation involve pre-cooked "templates" of Java, C++ or whatever the target language is, with spots for simple substitution of XML element contents. This situation is hard to avoid, especially in the context of *ns-3*. As described in [24], most code generation is done in the context of software engineering to enable the automatic generation of software components that would otherwise be tedious or difficult to write accurately by hand. Examples include GUI component creation, and middleware deployment [24].

In the case of *ns-3*, a single simulation script can hardly be considered the kind of boilerplate or repetitive code that is usually the target for code generation. Indeed, the code base for *ns-3* is one that is been under development for several years and is comprised of hundreds of thousands of lines of code. Frankly, it is unreasonable to imagine that "true" XML-to-C++ code generation could be implemented in the time span I have had without any templating "shortcut."

To this end, the code generation scheme I have formulated favors the re-use of existing *ns-3* scripts, similar to the way XSLT-style code generation re-uses code blocks. In my implementation, an *ns-3* script is made into a very sparse XML document, described by the NSTL Schema (see Appendix A.2. It consists of zero or more `group` elements which partition the script into segments. Within each `group`, one can insert any number of `block` tags to delimit blocks of code within the `group`. The idea is that if a researcher wishes to run a simulation using C++ object `ModelA` and compare it to the same simulation using object `ModelB`, they can write the necessary code used to construct each object within their own `block` element. During the process of code generation, a separate two scripts will be created; in the first, `ModelA` will be used and in the second, `ModelB` will be used.

Converting an existing script to an NSTL document is not difficult. In fact, simply adding an opening and closing `<template>` tag to the very first and last lines of an *ns-3* script will produce a valid NSTL document. Within the root `template` tag one delimits code segments with `group` tags; blocks of code within a `group` tag can be delimited with `block` tags.

7.3.3 How it Works

The code generation scheme works like this: each `group` element acts as a "placeholder" in the output script. If there are three `block` elements in a `group` then there will be three different output scripts generated. In the first, the text inside the first `block` will appear within the space held by the `group`; in the second, the text inside the second `block` element will appear within the space held by the `group`, and so on. To illustrate, consider the following small example in Listing 7.4.

```
1  public static void main(String[] args) {
2      System.out.println("This is the beginning of the main method!");
3      <group id="greeting">
4          <block>
5          System.out.println("Hello!");
6          </block>
7          <block>
8          System.out.println("Bonjour!");
9          </block>
10     </group>
11     <group id="farewell">
12         <block>
13         System.out.println("Goodbye!");
14         </block>
15         <block>
16         System.out.println("Auf Wiedersehen!");
17         </block>
18     </group>
19 }
```

Listing 7.4: An example Java method with XML additions.

In this example the highlighted XML dominates the line count, but this is only because such small blocks of code appear in each `block` element. The code generator will generate a total of four output scripts, since there are two `block` elements in the first `group`, two `block` elements in the second `group` and $2 \times 2 = 4$. The resulting scripts would be as follows:

```
1 public static void main(String[] args) {
2     System.out.println("This is the beginning of the main method.");
3     System.out.println("Hello!");
4     System.out.println("Goodbye!");
5 }
```

```
1 public static void main(String[] args) {
2     System.out.println("This is the beginning of the main method.");
3     System.out.println("Hello!");
4     System.out.println("Auf Wiedersehen!");
5 }
```

```
1 public static void main(String[] args) {
2     System.out.println("This is the beginning of the main method.");
3     System.out.println("Bonjour!");
4     System.out.println("Goodbye!");
5 }
```

```
1 public static void main(String[] args) {
2     System.out.println("This is the beginning of the main method.");
3     System.out.println("Bonjour!");
4     System.out.println("Auf Wiedersehen!");
5 }
```

Listing 7.5: Four sample output scripts.

The `group` and `block` elements allow for composition at the model level, allowing an experimenter to develop sophisticated code blocks and swap those out for other blocks of arbitrarily-complex code.

7.4 Chapter Summary

This chapter discussed the process of code generation within the SAFE framework and how NSTL constructs facilitate script templating. XML-to-C++ code generation is a long-term goal of interest to the *ns-3* community. An ideal scenario would be one in which novice and veteran users alike could quickly build up a set of models and

experimental parameter values for those models and observe the resulting simulation behavior. XML is well-suited for handling the input end of this architecture. It is a well-understood technology for which many libraries are available in every major language to handle. Already there exist graphical user interfaces that help automate the creation of XML documents. As it stands today, however, simple value substitution seems to be the most widely-used technique for code generation. Under this scheme, blocks of pre-cooked script code surround a special marker which is then automatically replaced by a hard-coded value.

My code generation scheme takes a similar approach, but does so without the use of XSLT. Instead, pre-existent C++ scripts are augmented with certain NSTL tags to turn them into a "template" document that validates against the NSTL Schema. Using `group` and `block` elements in combination with the value substitution markers (identified with `$` signs) a user can develop a sophisticated compositions of models and parameter values which are then transformed into many unique simulation scripts and written to separate output files.

Chapter 8

Conclusions and Future Work

This work began as a general investigation into network simulation, and especially the kind of complex, large-scale network simulation that is difficult to conduct. This difficulty is compounded by the lack of credibility in network simulation research, which can be attributed a number of reasons, including

- the large number of viable simulation engines available and the difficulty in replicating work done on one simulator with another;
- the increasing complexity of simulator engines as researchers find need to add custom modules and functionality to support their own investigations;
- the lack of complete documentation published to describe all of the decisions made in a simulation experiment (either explicitly by the researcher or implicitly because of default behaviors);
- the lack of a framework to guide less-than-expert users through the not only the simulation process itself, but the entire experiment design and execution phases.

This last point is especially important; intimate knowledge of a particular simulator engine does not necessarily imply knowledge of domain-specific best practices, experimental design, or even what qualifies as a sound scientific study. Perhaps researchers in the field of network simulation take this knowledge as a given. If so,

they have been disappointingly wrong; time and again published network simulation research has been shown to be at best incomplete and at worst scientifically unsound [27].

My work with Dr. Felipe Perrone has provided first steps towards the end-goal of simulator-independent experiment and model description. The goal is ambitious but frankly, infeasible for an undergraduate student in this space of time. Because of this, we have narrowed our scope to focus just on the *ns-3* network simulator. *ns-3* is a very powerful simulator used widely by researchers across the field. Our goal was to develop languages to support experiment and model description and integrate those languages into a larger experiment automation framework. This framework is named the Simulation Automation Framework for Experiments (SAFE). The experiment and model description languages used in this framework fit on top of the existing *ns-3* architecture and allow for a substantial (though not yet complete) degree of experiment automation.

From the start, we believed that creating XML-based experiment and model description languages was the best choice. XML is a well-established standard used by researchers, businesspeople and students in nearly every application domain. There exist a variety of tools that can be used to aid in the development of XML languages, as well as those for parsing, querying, validating and manipulating XML instance documents. Still, I have investigated other languages too, notably DML and ANML, and found that they are not as attractive as XML. XML is fine as an experiment description language, and though it possesses limitations as a model description language, those limitations are made up for in XML's wide support for document processing and validation.

The *ns-3* Experiment Description Language (NEDL) and *ns-3* Script Templating Language (NSTL) were the two languages I developed to serve the purposes of experiment description and model description, respectively. Experiment description is relatively straight-forward with NEDL and the language is designed to make the process intuitive to a novice. Special constructs in NEDL allow for efficient pruning of unwanted design points, which streamlines the experiment workflow. There are no constructs in NEDL that necessarily tie the language to description of *ns-3* experiments. Because its design was motivated by literature published in the field of experiment design (as opposed to the more specific field of network simulation experiment design), it could very well be used to describe any sort of experiment which pairs factors represented as strings to level values represented as either strings or numbers.

NSTL represents a step back from true model description and instead is intended to provide an easy templating mechanism for *ns-3* scripts. Given the complexity of many *ns-3* scripts, this solution is one way of allowing expert users to at least provide well-documented examples that others can easily manipulate and test using NSTL instead of changing the original script code itself. Like NEDL, there are no constructs in the NSTL language that necessarily tie the language to *ns-3*. Any text file, programming code or otherwise, could be augmented with the correct NSTL tags that are used in an NSTL document. One could generate a set of MATLAB scripts, for example, using NEDL and NSTL.

Document validation is a crucial piece of the entire SAFE framework and is carried out by checking NEDL and NSTL documents against the NEDL and NSTL Schemas, written in the W3C XML Schema language. Users have much greater responsibility in writing semantically-correct NSTL documents, since code blocks and groups are all assumed to be valid *ns-3* code. NEDL documents have a much more rigorous specification and are validated at a level of granularity that even the NEDL Schema cannot provide. To provide this validation, I have written a Python module called `NEDLValidator` that performs a series of checks to ensure that the document contents are semantically correct. `NEDLValidator` checks several properties of the document that the NEDL Schema cannot, such as comparing strings within the document against each other or against external filenames.

Code generation represents the final piece of my work within the SAFE framework. Given the deliberate design of NSTL as a templating language, code generation is an easily-automated process of using design points to generate an appropriate number of *ns-3* scripts using the `group` and `block` elements in the NSTL document. Given a valid NEDL file, certain modules in the SAFE framework can generate a series of experimental design points which are given to the code generating module I have implemented in Python. For each design point generated, this module generates the appropriate number of output scripts (depending the number of `block` elements within each `group`) and makes the proper strings replacements of model parameter placeholders with specific level values.

Future Work

NEDL and NSTL were designed with straight-forwardness in mind; they were kept simple in design to give both novice and expert users easy accessibility. It is not espe-

cially difficult to create and edit these documents by hand, though the development of front-end interfaces to documents is certainly an area of future work. The `lexpr` and `rexpr` expression elements in NSTL, for example, lend themselves naturally to automatic generation. An alternative design choice would have been to represent expressions as strings, such as "`i * 5 + 2`", rather than verbose XML element trees. But, pulling each part of the expression out into individual XML elements makes the development of interfaces that create and edit these expressions easier since there is no expression string parsing required.

Appendix A

Schemas

A.1 NEDL Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.bucknell.edu/safe/exp"
xmlns:tns="http://www.bucknell.edu/safe/exp"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:include schemaLocation="LevelList.xsd"/>
    <xs:element name="rvar"     type="xs:string"/>
    <xs:element name="rconst"   type="xs:string"/>
    <xs:element name="var"      type="xs:string"/>
    <xs:element name="lconst"   type="xs:string"/>
    <xs:element name="lvar"     type="xs:string"/>
    <xs:element name="lo"       type="xs:string"/>
    <xs:element name="hi"       type="xs:string"/>
    <xs:element name="delta"    type="xs:string"/>
    <xs:element name="factorid" type="xs:string"/>
    <xs:element name="test"     type="xs:string"/>
    <xs:element name="op"       type="xs:string"/>
    <xs:element name="value"    type="xs:string"/>
    <xs:element name="indexid"  type="xs:string"/>
    <xs:element name="listid"   type="xs:string"/>
```

```
21      <xs:element name="factor"   type="xs:string"/>
22
23      <xs:element name="experimentspace" type="tns:ExperimentSpace"/>
24      <xs:element name="factorlist" type="tns:FactorList"/>
25      <xs:element name="conditions" type="tns:Conditions"/>
26      <xs:element name="memberof" type="tns:MemberOf"/>
27      <xs:element name="sequence" type="tns:Sequence"/>
28      <xs:element name="lexpr" type="tns:Expression"/>
29      <xs:element name="rexpr" type="tns:Expression"/>
30      <xs:element name="where" type="tns:Where"/>
31      <xs:element name="range" type="tns:RangeConstraint"/>
32      <xs:element name="linkingrestriction"
33                  type="tns:LinkingRestriction"/>
34      <xs:element name="exclusionrestriction"
35                  type="tns:ExclusionRestriction"/>
36      <xs:element name="setting"
37                  type="tns:PointComponent"/>
38
39      <xs:complexType name="LinkingRestriction">
40          <xs:sequence>
41              <xs:element ref="tns:factor" maxOccurs="unbounded"/>
42          </xs:sequence>
43      </xs:complexType>
44
45      <xs:complexType name="FactorList">
46          <xs:sequence>
47              <xs:element ref="tns:factor" maxOccurs="unbounded"/>
48          </xs:sequence>
49      </xs:complexType>
50
51      <xs:complexType name="ExperimentSpace">
52          <xs:sequence>
53              <xs:element ref="tns:factorlist"/>
54              <xs:element ref="tns:conditions"/>
55          </xs:sequence>
56      </xs:complexType>
57
58      <xs:complexType name="PointComponent">
59          <xs:attribute name="factor" type="xs:string"/>
```

```
60          <xs:attribute name="level" type="xs:string"/>
61      </xs:complexType>
62
63      <xs:complexType name="MemberOf">
64          <xs:sequence>
65              <xs:element ref="tns:factor"/>
66              <xs:element ref="tns:listid"/>
67          </xs:sequence>
68      </xs:complexType>
69
70      <xs:complexType name="Conditions">
71          <xs:sequence>
72              <xs:element ref="tns:memberof"
73                          maxOccurs="unbounded"
74                          minOccurs="0"/>
75              <xs:element ref="tns:levellist"
76                          maxOccurs="unbounded"
77                          minOccurs="0"/>
78              <xs:element ref="tns:sequence"
79                          maxOccurs="unbounded"
80                          minOccurs="0"/>
81              <xs:element ref="tns:exclusionrestriction"
82                          maxOccurs="unbounded"
83                          minOccurs="0"/>
84              <xs:element ref="tns:linkingrestriction"
85                          maxOccurs="unbounded"
86                          minOccurs="0"/>
87          </xs:sequence>
88      </xs:complexType>
89
90      <xs:complexType name="Sequence">
91          <xs:sequence>
92              <xs:element ref="tns:factor"/>
93              <xs:element ref="tns:test"/>
94              <xs:choice>
95                  <xs:element ref="tns:lexpr"/>
96                  <xs:element ref="tns:lvar"/>
97                  <xs:element ref="tns:lconst"/>
98              </xs:choice>
```

```
99              <xs:element ref="tns:op"/>
100             <xs:choice>
101                 <xs:element ref="tns:rexpr"/>
102                 <xs:element ref="tns:rvar"/>
103                 <xs:element ref="tns:rconst"/>
104             </xs:choice>
105             <xs:element ref="tns:where"
106                         maxOccurs="1"
107                         minOccurs="0"/>
108         </xs:sequence>
109     </xs:complexType>
110
111     <xs:complexType name="Where">
112         <xs:sequence>
113             <xs:element ref="tns:range"/>
114         </xs:sequence>
115     </xs:complexType>
116
117     <xs:complexType name="RangeConstraint">
118         <xs:sequence>
119             <xs:element ref="tns:var"/>
120             <xs:element ref="tns:lo"/>
121             <xs:element ref="tns:hi"/>
122             <xs:element ref="tns:delta"/>
123         </xs:sequence>
124     </xs:complexType>
125
126     <xs:complexType name="Expression">
127         <xs:sequence>
128             <xs:element ref="tns:lexpr"
129                         maxOccurs="1"
130                         minOccurs="0"/>
131             <xs:element ref="tns:lvar"
132                         maxOccurs="1"
133                         minOccurs="0"/>
134             <xs:element ref="tns:lconst"
135                         maxOccurs="1"
136                         minOccurs="0"/>
137             <xs:element ref="tns:op"/>
```

```
138             <xs:element ref="tns:rexpr"
139                         maxOccurs="1"
140                         minOccurs="0"/>
141             <xs:element ref="tns:rvar"
142                         maxOccurs="1"
143                         minOccurs="0"/>
144             <xs:element ref="tns:rconst"
145                         maxOccurs="1"
146                         minOccurs="0"/>
147         </xs:sequence>
148     </xs:complexType>
149 
150     <xs:complexType name="ExclusionRestriction">
151         <xs:sequence>
152             <xs:element ref="tns:setting" maxOccurs="unbounded"/>
153         </xs:sequence>
154     </xs:complexType>
155 
156 </xs:schema>
```

A.2 NSTL Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema targetNamespace="http://www.bucknell.edu/safe/exp"
3  xmlns:tns="http://www.bucknell.edu/safe/exp"
4  xmlns:xs="http://www.w3.org/2001/XMLSchema">
5  
6      <xs:element name="template" type="tns:Template"/>
7      <xs:element name="group" type="tns:Group"/>
8      <xs:element name="block" type="xs:string"/>
9  
10     <xs:complexType name="Template" mixed="true">
11         <xs:sequence>
12             <xs:element ref="tns:group"
13                         minOccurs="0"
14                         maxOccurs="unbounded"/>
15         </xs:sequence>
```

```
16      </xs:complexType>
17
18      <xs:complexType name="Group">
19          <xs:sequence>
20              <xs:element ref="tns:block"
21                          minOccurs="0"
22                          maxOccurs="unbounded"/>
23          </xs:sequence>
24          <xs:attribute name="id"/>
25      </xs:complexType>
26
27  </xs:schema>
```

A.3 LevelList Schema

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <xs:schema targetNamespace="http://www.bucknell.edu/safe/exp"
3   xmlns:tns="http://www.bucknell.edu/safe/exp"
4   xmlns:xs="http://www.w3.org/2001/XMLSchema">
5
6           <xs:element name="levellist" type="tns:LevelList"/>
7       <xs:element name="level" type="xs:string"/>
8
9           <xs:complexType name="LevelList">
10                  <xs:sequence>
11              <xs:element ref="tns:level"
12                          maxOccurs="unbounded"
13                          minOccurs="0"/>
14                  </xs:sequence>
15          <xs:attribute name="id" type="xs:string"/>
16          </xs:complexType>
17
18  </xs:schema>
```

References

[1] PyXB Python XML Schema Bindings. Available at `http://pyxb.sourceforge.net/index.html`. [Accessed September 6, 2010].

[2] ISO Schematron. Available at `http://www.schematron.com` [Access September 6, 2010].

[3] *Domain Modeling Language (DML) Reference Manual*, 1999. Available at `http://www.ssfnet.org/SSFdocs/dmlReference.html`. [Accessed April 3, 2011].

[4] The ns-3 network simulator, 2011. Available at `http://www.nsnam.org/index.html`. [Accessed March 29, 2011].

[5] Scalable simulation framework, 2002. Available at `http://www.ssfnet.org/homePage.html`. [Accessed April 3, 2011].

[6] World wide web consortium (W3C). Available at `http://www.w3.org/`. [Accessed April 4, 2011].

[7] Michele Amoretti, Matteo Agosti, and Francesco Zanichelli. DEUS: a discrete event universal simulator. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 58:1–58:9, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-45-5. doi: http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5754. URL `http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5754`.

[8] Perakath Benjamin, Mukul Patki, and Richard Mayer. Using ontologies for simulation modeling. In *Proceedings of the 38th Winter Simulation Conference*, WSC '06, pages 1151–1159. Winter Simulation Conference, 2006. ISBN 1-4244-0501-7. URL `http://portal.acm.org/citation.cfm?id=1218112.1218321`.

[9] M. Bertoli, G. Casale, and G. Serazzi. The JMT simulator for performance evaluation of non-product-form queueing networks. In *Proceedings of the 40th Annual Simulation Symposium*, pages 3–10, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2814-7. doi: 10.1109/ANSS.2007.41. URL `http://portal.acm.org/citation.cfm?id=1249253.1250388`.

[10] James M. Brase and David L. Brown. Modeling, simulation and analysis of complex networked systems. Technical report, Lawrence Livermore National Laboratory, May 2009.

[11] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. *XML Version 1.0*, 2008. Available at `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[12] Don Brutzman. XMSF: Extensible modeling and simulation framework, 2004. Available at `https://www.movesinstitute.org/xmsf/xmsf.html`. [Accessed March 29, 2011].

[13] R. Canonico, D. Emma, and G. Ventre. An XML description language for web-based network simulation. In *Distributed Simulation and Real-Time Applications, 2003. Proceedings. Seventh IEEE International Symposium on*, pages 76–81, Oct. 2003.

[14] D Carlisle, P. Ion, R. Miner, and N. Poppelier. *MathML Version 2.0*, 2003. Available at `http://www.w3.org/TR/MathML2/`.

[15] J. Clark. *XML Transformations (XSLT) version 1.0*, 1999. Available at `http://www.w3.org/TR/xslt`.

[16] J. Clark and M. Murata. *RELAX NG Specification*, 2001. Available at `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[17] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[18] Andrea D'Ambrogio. A model transformation framework for the automated building of performance models from UML models. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 75–86, New York, NY, USA, 2005. ACM. ISBN 1-59593-087-6. doi: http://doi.acm.org/10.1145/1071021.1071029. URL `http://doi.acm.org/10.1145/1071021.1071029`.

[19] Andrea D'Ambrogio. A model-driven WSDL extension for describing the QoS of web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 789–796, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2669-1. doi: 10.1109/ICWS.2006.10. URL `http://portal.acm.org/citation.cfm?id=1172963.1173139`.

[20] Paul A. Fishwick and John A. Miller. Ontologies for modeling and simulation: issues and approaches. In *Proceedings of the 36th Winter Simulation Conference*, WSC '04, pages 259–264. Winter Simulation Conference, 2004. ISBN 0-7803-8786-4. URL `http://portal.acm.org/citation.cfm?id=1161734.1161788`.

[21] Steve Franklin. Xml parsers: DOM and SAX put to the test. `http://www.devx.com/xml/Article/16922/1954`.

[22] John Grundy, Yuhong Cai, and Anna Liu. SoftArch/MTE: Generating distributed system test-beds from high-level software architecture descriptions. *Automated Software Engg.*, 12:5–39, January 2005. ISSN 0928-8910. doi: 10.1023/B:AUSE.0000049207.62380.74. URL `http://portal.acm.org/citation.cfm?id=1035394.1035410`.

[23] Koichi Hayashi and Riichiro Mizoguchi. Document exchange model for augmenting added value of B2B collaboration. In *Proceedings of the 5th International Conference on Electronic Commerce*, ICEC '03, pages 458–464, New York, NY, USA, 2003. ACM. ISBN 1-58113-788-5. doi: http://doi.acm.org/10.1145/948005.948064. URL `http://doi.acm.org/10.1145/948005.948064`.

[24] Cody Henthorne and Eli Tilevich. Code generation on steroids: Enhancing COTS code generators via generative aspects. In *Proceedings of the Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques*, IWICSS '07, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2966-6. doi: http://dx.doi.org/10.1109/IWICSS.2007.4. URL `http://dx.doi.org/10.1109/IWICSS.2007.4`.

[25] C. Kiddle, R. Simmonds, D.K. Wilson, and B. Unger. ANML: A language for describing networks. In *Proceedings of the Ninth Internation Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)*, pages 135 –141, 2001. doi: 10.1109/MASCOT.2001.948862.

[26] Dagmar Köhn and Nicolas Novère. SED-ML – an XML format for the implementation of the MIASE guidelines. In *Proceedings of the 6th International Conference on Computational Methods in Systems Biology*, CMSB '08,

pages 176–190, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88561-0. doi: http://dx.doi.org/10.1007/978-3-540-88562-7_15. URL `http://dx.doi.org/10.1007/978-3-540-88562-7_15`.

[27] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. MANET simulation studies: the incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):50–61, 2005. ISSN 1559-1662. doi: http://doi.acm.org/10.1145/1096166.1096174.

[28] Ivan Kurtev and Klaas van den Berg. Building adaptable and reusable XML applications with model transformations. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 160–169, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9. doi: http://doi.acm.org/10.1145/1060745.1060772. URL `http://doi.acm.org/10.1145/1060745.1060772`.

[29] Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill, 4th edition, 2007. ISBN 0072988436.

[30] Jason Liu, L. Felipe Perrone, David M. Nicol, Michael Liljenstam, Chip Elliot, and David Pearson. Simulation modeling of large-scale ad-hoc sensor networks. In *Proceedings of the 2001 European Simulation Interoperability Workshop*, London, England, 2001.

[31] John A. Miller, Gregory T. Baramidze, Amit P. Sheth, and Paul A. Fishwick. Investigating ontologies for simulation modeling. In *Proceedings of the 37th Annual Symposium on Simulation*, ANSS '04, pages 55–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2110-X. URL `http://portal.acm.org/citation.cfm?id=987679.987721`.

[32] Max D. Morris. *Design of Experiments: An Introduction Based on Linear Models*. Chapman & Hall/CRC, 2011.

[33] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, 2005. ISSN 1533-5399. doi: http://doi.acm.org/10.1145/1111627.1111631.

[34] Peter Murray-Rust and Henry S. Rzepa. *Chemical Markup Language. A Position Paper*, 2001. Available at `http://cml.sourceforge.net/historical/position.html`. [Accessed March 9, 2011].

[35] David M. Nicol, Michael Liljenstam, and Jason Liu. Advanced concepts in large-scale network simulation. In *Proceedings of the 37th Winter Simulation Conference*, WSC '05, pages 153–166. Winter Simulation Conference,

2005. ISBN 0-7803-9519-0. URL `http://portal.acm.org/citation.cfm?id=1162708.1162740`.

[36] Matthias Nicola and Jasmi John. XML parsing: a threat to database performance. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 175–178, New York, NY, USA, 2003. ACM. ISBN 1-58113-723-0. doi: http://doi.acm.org/10.1145/956863.956898. URL `http://doi.acm.org/10.1145/956863.956898`.

[37] *The ns-3 network simulator*. ns-3. Available at `http://www.nsnam.org/`.

[38] Daisuke Okajima and Kohsuke Kawaguchi. *RelaxNGCC (RelaxNG Compiler Compiler)*. Available at `http://relaxngcc.sourceforge.net/en/index.htm`. [Accessed March 18, 2011].

[39] Kara A. Olson, C. Michael Overstreet, and E. Joseph Derrick. Code analysis and CS-XML. In *Proceedings of the 39th Winter Simulation Conference*, WSC '07, pages 756–761, Piscataway, NJ, USA, 2007. IEEE Press. ISBN 1-4244-1306-0. URL `http://portal.acm.org/citation.cfm?id=1351542.1351681`.

[40] L. Felipe Perrone, Christopher J. Kenna, and Bryan C. Ward. Enhancing the credibility of wireless network simulations with experiment automation. In *WIMOB '08: Proceedings of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication*, pages 631–637, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3393-3. doi: http://dx.doi.org/10.1109/WiMob.2008.53.

[41] L. Felipe Perrone, Claudio Cicconetti, Giovanni Stea, and Bryan C. Ward. On the automation of computer network simulators. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-45-5. doi: http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5684.

[42] Luiz Felipe Perrone and David M. Nicol. A scalable simulator for TinyOS applications. In *Proceedings of the 34th Winter Simulation Conference (WSC '02)*, pages 679–687. Winter Simulation Conference, 2002. ISBN 0-7803-7615-3.

[43] Steven W. Reichenthal. Srml case study: simple self-describing process modeling and simulation. In *Proceedings of the 36th Winter Simulation Conference*, WSC '04, pages 1461–1466. Winter Simulation Conference, 2004. ISBN 0-7803-8786-4. URL `http://portal.acm.org/citation.cfm?id=1161734.1162004`.

[44] F. Rioux, F. Bernier, and D. Laurendeau. Design and implementation of an XML-based, technology-unified data pipeline for interactive simulation. In *Simulation Conference, 2008. WSC 2008. Winter*, pages 1130–1138, Dec. 2008. doi: 10.1109/WSC.2008.4736182.

[45] M. Röhl and A.M. Uhrmacher. Composing simulations from XML-specified model components. In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 1083–1090, Dec. 2006. doi: 10.1109/WSC.2006.323198.

[46] Susan M. Sanchez. Work smarter, not harder: guidelines for designing simulation experiments. In *WSC '06: Proceedings of the 38th Winter Simulation Conference*, pages 47–57. Winter Simulation Conference, 2006. ISBN 1-4244-0501-7.

[47] C.M. Sperberg-McQueen and Henry Thompson. *W3C XML Schema*. W3C, 2007. Available at `http://www.w3.org/XML/Schema`.

[48] Galen S. Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai, and Koichi Moriyama. Clearwater: extensible, flexible, modular code generation. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 144–153, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: http://doi.acm.org/10.1145/1101908.1101931. URL `http://doi.acm.org/10.1145/1101908.1101931`.

[49] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*, 2001. Available at `http://www.w3.org/TR/xmlschema-1/`.

[50] James R. Wilson. Conduct, misconduct, and cargo cult science (doctoral colloquium keynote address). In *Proceedings of the 29th Winter Simulation Conference*, WSC '97, pages 1405–1413, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-7803-4278-X. doi: http://dx.doi.org/10.1145/268437.268790. URL `http://dx.doi.org/10.1145/268437.268790`.