The ANML Guide

Cameron Kiddle

TeleSim Group
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, AB T2N 1N4, Canada

January 23, 2002

Contents

1 Introduction **3**
1.1 Origin . . . 3
1.2 Design Goals . . . 3
1.3 Terminology . . . 3
1.4 Notation . . . 4
1.5 Document Layout . . . 5

2 Tutorial: How to Define an ANML Schema **6**
2.1 Step 1 - Establish the General Structure of What is to be Modelled . . . 6
2.2 Step 2 - Write the Schema . . . 7
2.2.1 Defining a Class . . . 7
2.2.1.1 Declaring a Valid Sub-Component . . . 8
2.2.1.2 Defining a Class Attribute . . . 9
2.2.1.3 Assigning a Default Attribute Value . . . 10
2.2.2 Declaring a Valid Top Level Component . . . 11

3 Tutorial: How to Create an ANML Model **12**
3.1 Step 1 - Decide What to Model . . . 12
3.2 Step 2 - Break the Model Down into Individual Components . . . 12
3.3 Step 3 - Define Components Needed by the Model in a Database . . . 13
3.4 Step 4 - Define the Model . . . 19

4 A Formal Definition of ANML **20**
4.1 ANML Syntax . . . 20
4.2 Well-Formedness and Validity . . . 21
4.3 Model and Key-Value Pairs . . . 21
4.4 Keys . . . 21
4.4.1 Component Keys . . . 21
4.4.1.1 Component Definitions . . . 22
4.4.1.2 Component Instances . . . 22
4.4.2 Attribute Keys . . . 22
4.4.3 Reserved Keys . . . 23
4.4.3.1 _CLASS_KEY . . . 23
4.4.3.2 _DATABASE_KEY . . . 23
4.4.3.3 _FROM_KEY . . . 24
4.4.3.4 _ID_KEY . . . 24
4.4.3.5 _IN_DATABASE_KEY . . . 24
4.4.3.6 _INCLUDE_KEY . . . 25
4.4.3.7 _MODEL_KEY . . . 25
4.4.3.8 _NAME_KEY . . . 25
4.4.3.9 _TO_KEY . . . 26
4.4.3.10 _USE_SCHEMA_KEY . . . 26
4.5 Values . . . 26
4.6 Comments . . . 27
4.7 Identification System . . . 28

5 A Formal Definition of the ANML Schema **29**
5.1 ANML Schema Syntax . . . 29
5.2 Schema and Schema Specifications . . . 30
5.3 Components and Component Specifications . . . 30
5.4 Classes and Class Specifications . . . 31
5.5 Attributes and Attribute Specifications . . . 32
5.6 Default Values . . . 33

5.7 Component Occurrence Constraints 34
5.7.1 _NUM_OCCUR_KEY 34
5.7.2 _MIN_OCCUR_KEY 34
5.7.3 _MAX_OCCUR_KEY 35
5.8 Attribute Value Constraints 35
5.8.1 _LENGTH_KEY 36
5.8.2 _MIN_LENGTH_KEY 36
5.8.3 _MAX_LENGTH_KEY 36
5.8.4 _NUM_ENTRIES_KEY 37
5.8.5 _MIN_ENTRIES_KEY 37
5.8.6 _MAX_ENTRIES_KEY 38
5.8.7 _NUM_IDS_KEY 38
5.8.8 _MIN_IDS_KEY 38
5.8.9 _MAX_IDS_KEY 39
5.8.10 _MIN_INCLUSIVE_KEY 39
5.8.11 _MIN_EXCLUSIVE_KEY 39
5.8.12 _MAX_INCLUSIVE_KEY 40
5.8.13 _MAX_EXCLUSIVE_KEY 41
5.8.14 _VALID_CLASSES_KEY 41
5.8.15 _ONE_OF_KEY 41
5.8.16 _PATTERN_KEY 42
5.9 Attribute Types 42
5.9.1 Primitive Attribute Types 42
5.9.1.1 INTEGER_TYPE 43
5.9.1.2 REAL_TYPE 43
5.9.1.3 BOOLEAN_TYPE 43
5.9.1.4 STRING_TYPE 44
5.9.1.5 ID_TYPE 44
5.9.2 Compound Attribute Types 44
5.9.2.1 INTEGER_LIST_TYPE 45
5.9.2.2 REAL_LIST_TYPE 45
5.9.2.3 BOOLEAN_LIST_TYPE 46
5.9.2.4 STRING_LIST_TYPE 46
5.9.2.5 COMP_ATR_TYPE 46

6 Processing ANML Models **48**

A ANML Files Used in Tutorials **49**

1 Introduction

Another Modelling Language (ANML) is a language that has been created for describing communication network models. However, ANML has been designed to describe other types of models as well. A structure called a schema can be defined to create a new model type. Schemas specify the types of components that can be contained in a model, the attributes associated with the components, and how components can be composed together. A language called the ANML Schema language is used to define a schema.

1.1 Origin

ANML resulted from a project, started in the fall of 1999, to develop a configuration language for the *Internet Protocol Traffic and Network* (IP-TN) simulator. IP-TN was developed by the TeleSim group at the University of Calgary to complement an existing Asynchronous Transfer Mode (ATM) network simulator called the *ATM Traffic and Network*(ATM-TN) simulator [8]. ATM-TN has a complex configuration interface, and thus a new and easier configuration interface for IP-TN was desired.

Two languages served as a basis for ANML. One of these languages was the *Domain Modelling Language* (DML) [7]. The origins of DML are in the Graph Modelling Language (GML) [4, 5] which is a language for describing graphs. DML has been used to configure network scenarios for SSFNET [1, 2, 3], a network simulator developed by the *Scalable Simulation Framework* (SSF) research group. The other language was the *Extensible Markup Language* (XML) [9]. XML provides a common format for sharing structured information over the Internet. DML and XML both use schemas to the types of structures or models that can be created. ANML derives its syntax from DML and its logical structure from XML. Several features of the ANML Schema language are derived from the XML Schema language.

1.2 Design Goals

The design goals for ANML are given below:

- **Simple:** Models should be able to be created easily.
- **Legible:** ANML documents should be able to be read and understood by humans.
- **Terse:** A model should be able to be described in a concise manner.
- **Hierarchical:** Hierarchical structures should be able to be represented in ANML.
- **Composable:** Larger components should be able to be constructed using smaller components. These components should be able to be archived for reuse.
- **Extensible:** New component types should be able to be added.
- **Verifiable:** Models should be able to be verified for correct composure and description.
- **Position Independent:** Data should be able to be placed in any order.
- **Unique Identification:** All elements in a model should be able to be uniquely identified.

1.3 Terminology

- **model definition:** an overall set of ANML documents that contain the schema, databases and component instances to describe a model
- **schema definition:** given in a model definition to define the structure and compositional rules for the model
- **database definition:** given in a model definition to create an archive of components that can be used to create models
- **class definition:** given in a schema definition to specify a component type
- **component declaration:** given in a class definition to indicate that it is an allowable sub-component of the class

- **attribute definition:** given in a class definition to specify an attribute associated with the component class along with its attribute type and value constraints
- **constraint definition:** given in a component declaration or attribute definition to restrict the number of component occurrences or the value of an attribute
- **component definition:** given in a database definition to create a component with specific sub-components and attributes for later re-use
- **component instance:** given in a component definition, component instance or model definition to create an actual component that exists in the model
- **attribute assignment:** given in a component definition or a component instance to assign a value to an associated attribute of the component
- **key:** a string of symbols that is used as an identifier for a component, attribute or other reserved feature
- **value:** represent information pertaining to a key
- **key-value pair:** a key and an associated value which provides information pertaining to the key
- **well-formedness constraint:** a rule that must be satisfied by a model, in order for the model to be considered well-formed.
- **validity constraint:** a rule that must be satisfied by a model, in order for the model to be considered valid.
- **well-formed:** an ANML model is considered to be well-formed if it matches the ANML syntax and any additional well-formedness constraints
- **valid:** an ANML model is considered to be valid if it satisfies the constraints and structure imposed by the schema definition and any additional validity constraints
- **ANML Processor:** a software module that reads in an ANML model and checks for well-formedness and validity of the model
- **application:** a software module that inputs and utilizes an ANML model that has been processed by the ANML Processor

1.4 Notation

The grammars for ANML and the ANML Schema are presented in a manner similar to that of the Extended Backus-Naur Form (EBNF). A grammar is represented by a set of rules called productions. Each production defines a symbol as shown below:

Production Notation		
symbol	::=	alternative 1 | alternative 2 | ... | alternative N

If more than one production is associated with a single symbol the '|' character is used to separate the alternative productions.

A symbol is either a terminal or non-terminal symbol. Terminal symbols are also referred to as tokens or a unit of the grammar that is indivisible. A terminal can be given directly as a literal string inside of quotes, or written using all upper case letters. Terminal symbols written in upper case are then defined in a production using a regular expression. Non-terminal symbols are written in all lower case letters and are defined in a production as a sequence of non-terminal and terminal symbols.

A regular expression is used to describe a certain pattern of characters. In this document, regular expressions are presented in the format used by *lex* [6]. Following is an explanation of the lex regular expression notations that are used in this document, as well as some examples.

Regular Expression Notation		
?	-	matches zero or one occurrence of the preceding expression
*	-	matches zero or more occurrences of the preceding expression
+	-	matches one or more occurrences of the preceding expression
[ ]	- - -	matches any character in brackets if first character in brackets is ‘ˆ’, matches any character except those inside the brackets a ‘-’ indicates a character range, except when it occurs as the first character in the brackets, in which case it indicates a dash itself i.e. [0-9] for all digits
( )	-	used to group a series of regular expressions together
“...”	-	everything in between quotes is interpreted literally
\	-	used to escape meta-characters or for C escape sequences i.e. ‘\?’ is a ‘?’, ‘*’ is a ‘*’, ‘\\’ is a ‘\’ i.e. ‘\n’ is a newline character , ‘\t’ is a tab character

Regular Expression Examples		
[0-9]+	-	matches one or more digits (i.e. a positive integer)
[-+]?[0-9]+	-	matches one or more digits that may begin with a sign (i.e. an integer)
[-+]?[0-9]+(\.[0-9]+)?	-	matches a decimal number where the decimal part is optional
[A-Z][a-z]*	-	matches a string that begins with an upper case letter and is followed by zero or more lower case letters
[ˆ \n\t]	-	matches any character except spaces, newlines and tabs
“?”	-	matches a question mark

1.5 Document Layout

The remainder of document proceeds as follows. Sections 2 and 3 present tutorials on how to define schemas and models. Formal definitions of the ANML and ANML Schema languages are then given in Sections 4 and 5 and are intended as reference manuals. A discussion of how ANML models are processed then follows in Section 6. An appendix is attached at the end of the document with the complete versions of the ANML documents used in the tutorials.

2 Tutorial: How to Define an ANML Schema

This tutorial describes the process to follow when defining an ANML Schema. A schema defines the allowable components, the associated attributes and the compositional rules of what is to be modelled. Application developers, or others who may be responsible for defining a schema, are intended as the primary audience of this section. However, general users of the application need to have a basic understanding of what is contained in a schema definition, as it is the schema that dictates what may be modelled.

There are two main steps in defining an ANML schema. The first step establishes the general structure of what is to be modelled and the second step is writing the schema. To help illustrate these steps, portions of an example schema for communication network models are presented and explained. A full version of the example schema can be found in appendix A. General users should read the second step.

2.1 Step 1 - Establish the General Structure of What is to be Modelled

The first step in defining a schema is establishing the general structure of what is to be modelled. This involves establishing the types of components that a model may be composed of, as well as determining how these components may be composed together. For example, if networks are to be modelled it is necessary to establish the components a network may contain and how they can be composed together to construct networks.

To start, an abstract description of what is to be modelled must be developed. Using the abstract description, the component types needed in the model can be identified and represented in a class structure. ANML has been designed to follow object oriented programming concepts meaning it has class hierarchy and inheritance. All classes must be derived from a built-in base class called ‘Component’.

After establishing a class structure for the component types, the compositional rules for the model components need to be identified. First, identify the different types of top level or main components. Then identify the types of sub-components the top level components can be composed of. This process is continued until the sub-component types cannot be further decomposed.

As an example, let us establish a possible structure for describing simple communication network models. First, an abstract description of a communication network model must be developed. Assume that we are only concerned with modelling the topology of the network. We are primarily interested in modelling wide area networks and are not too concerned about details of the exact topology at the local area network level. End nodes on the network should be capable of generating and receiving traffic and intermediate nodes should be capable of routing traffic to its correct destination. Let us call the end nodes hosts, and the intermediate nodes routers. Nodes are joined together by links.

From the abstract description we can see that there are three main types of components: networks, nodes and links. Two types of nodes need to be modelled, namely, hosts and routers. The types of links to model also need to be considered. We may have links that directly join two nodes together. Let us call these links point-to-point links or P2P links. We also need to represent the connection of multiple nodes on a local area network or LAN. LANs may be organized in topologies such as rings or stars, but since we are not concerned with the exact topology of LANs, we will model LANs with a single link to which all nodes on the LAN are attached. These will be called LAN links. The types mentioned above can be represented in a class structure for ANML as can be seen in Figure 1.

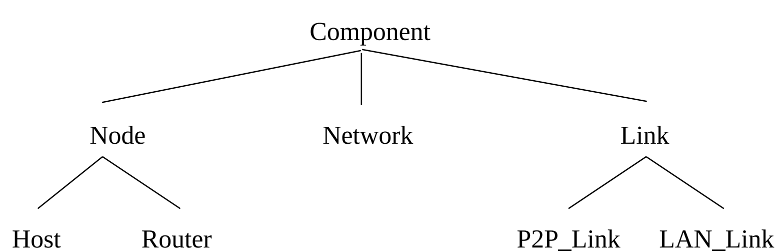

Figure 1: Network Schema Class Structure

Derived from the base ‘Component’ class are the ‘Network’, ‘Node’ and ‘Link’ classes. The ‘Router’ and ‘Host’ classes are derived from the ‘Node’ class to represent the two different types of nodes. The ‘P2P_Link’ (point-to-point links) and ‘LAN_Link’ classes are derived from the ‘Link’ class to represent the two different types of links. Should we wish to model additional types of nodes or links in the future we can easily add new sub-classes to the ‘Node’ and ‘Link’ classes.

Now let us establish the compositional rules. The main type of component that we are trying to model is a network. Networks consist of nodes and links to connect the nodes together. Also, networks are organized in a hierarchical manner, that is, they can be divided into sub-networks. The Internet is an example of a very large network that is composed of many different networks, each network being further divided into sub-networks and so on. Nodes and links may not be composed of any other components so we have discovered all of the compositional rules. A model may consist of networks and networks may consist of networks (sub-networks), nodes and links.

2.2 Step 2 - Write the Schema

Now that the class structure for the ANML schema has been defined, the schema can be written into an ANML file. It is best to define a schema in its own file as the schema will be used by many different model instances. The schema file can then be included in any of the model files. A schema is defined using the '_schema' reserved key. The general syntax for a schema is given below.

```
_schema [
  _name STRING      // name to identify the schema

  _classes [     // definition of component types
    Class1 []
    Class2 []
    ...
  ]

  _components [    // declaration of components valid at top level of model
    Component1 []
    Component2 []
    ...
  ]

]
```

A name must be given first to identify the schema using the '_name' reserved key. The name key may be assigned any value that matches the 'STRING' production of the ANML syntax given in Figure 9. For example, 'NetSchema' could be given as the name of the schema. After giving a name, the different types of components that may exist in a model need to be defined using the reserved key '_classes'. Each component type is represented by a class in ANML. Classes are defined as a sequence of key-value pairs, with each key being the name of a class. The types of components that may occur in the top level of a model are then declared using the reserved key '_components'. They are declared as a sequence of key-value pairs, with each key being the class name of a valid top level component. The '_classes' and '_components' sections may be placed in any order. As the '_components' section is normally quite small, it is suggested that it be placed before the '_classes' section to allow for easier reference to it. This section proceeds by showing how to define a class and how to declare a valid top level component type.

2.2.1 Defining a Class

The general syntax for defining a class is given below.

```
COMPONENT_NAME [  // the name of the class
  _isa COMPONENT_NAME // the name of the parent class - must be given first
  _app_class STRING   // name of the corresponding application class
  _may_instantiate BOOLEAN // indicates if class can be instantiated
  _components [ ... ] // declaration of allowable sub-components
  _attributes [ ... ] // definition of attributes associated with class
  _default [ ... ] // assignment of default attribute values
]
```

In defining a class, the name of the class is given first. The name of the class must match the 'COMPONENT_NAME' production of the ANML syntax given in Figure 9. Class specifications are then given as a sequence of key-value pairs. The reserved key '_isa' must be given first to indicate the parent class. The class given must be a class that is defined in the '_classes' section of the schema or it must be the 'Component' class. The 'Component' class is the base class of all classes defined in the schema. A class will inherit all of the component declarations, attribute definitions and default attribute values of its parent class and so on up the class hierarchy. The remaining class specifications may be given in any order and are all optional.

The '_app_class' reserved key can be used to specify the name of the corresponding class in an application using ANML models as input. This information can be used by the application to identify the type of object to build. For example, consider the 'Host' class in the network schema class structure given in Figure 1. An application could have a class called 'host_t' that represents a 'Host' in an ANML model. The '_app_class' reserved key could be assigned the value 'host_t'. Applications could also recognize the type of object to build by the name of the ANML class. In this case, the '_app_class' key-value pair does not need to be given. The use of this optional specification gives more freedom for creating or changing names of classes in an ANML schema or in an application.

Quite often it is desirable to have certain base classes from which others may be derived but are meaningless to instantiate on their own. For example, consider the 'Node' class in the network schema class structure given in Figure 1. 'Node' is a base class for the 'Host' and 'Router' classes. When instantiating a node we want to instantiate either a 'Host' or a 'Router'. Creating an instance of the 'Node' class would make no sense as we would not know the type of the node. The '_may_instantiate' reserved key allows one to specify whether or not instances of the class are allowed. By default, all classes are instantiable, so this specification need only be given if the class is not instantiable. To do this 'false' must be assigned as the 'BOOLEAN' value. The formal syntax for 'BOOLEAN' can be found in Section 5.9.1.3.

The remaining three class specifications are '_components', '_attributes' and '_default'. The '_components' reserved key is used to declare allowable sub-components of instances of this class. Any descendant classes of the declared sub-components are allowable as well. The '_attributes' reserved key is used to define any attributes associated with the class and the '_default' reserved key is used to assign default attribute values. Instructions on declaring valid sub-components, defining attributes and assigning default attribute values are given in Sections 2.2.1.1, 2.2.1.2 and 2.2.1.3 respectively.

As examples, let us define the 'Network', 'Node' and 'Router' classes in the network schema class structure given in Figure 1.

```
Network [
  _isa Component
  ...
]

Node [
  _isa Component
  _may_instantiate false
]

Router [
  _isa Node
  ...
]
```

The 'Network' and 'Node' classes are derived from the base 'Component' class whereas the 'Router' class is derived from the 'Node' class. We are not concerned with a corresponding application class and thus omit the '_app_class' reserved key. We want the 'Network' and 'Router' classes to be instantiable so we omit the '_may_instantiate' reserved key since by default all classes are instantiable. On the other hand, we do not want to allow instances of the 'Node' class so we give the '_may_instantiate' reserved key and assign it a value of 'false'. The remaining details of the 'Network' and 'Router' classes are given in the sub-sections below.

2.2.1.1 Declaring a Valid Sub-Component

Valid sub-components may be declared in the '_components' section of the class definition. The outline for declaring a valid sub-component is given below.

```
COMPONENT_NAME [ // The class name of the valid sub-component type
  _occurs [ ... ] // Occurrence constraints
]
```

An allowable sub-component is declared by first giving the class name as the key. As the value, occurrence constraints may or may not be specified. Occurrence constraints restrict the number of times a component of this class may occur as a sub-component of the class being defined. By default, one may have an unlimited number of occurrences of a component. A description of the different kinds of occurrence constraints can be found in Section 5.7.

The component declarations for the 'Network' class are given below.

```
      _components[
        /* A Network may contain subnets */
        Network []

        /* A Network may contain different nodes */
        Node []

        /* A Network may contain links to connect the nodes together */
        Link []

      ] // end Network _components
```

Networks may contain nodes and links and subnets which in turn may contain nodes and links and further subnets. We do not wish to constrain the allowable occurrences and thus do not specify any occurrence constraints. A router may not contain any sub-components and thus we do not define a '_components' section for the 'Router' class.

2.2.1.2 Defining a Class Attribute

Attributes may be defined in the '_attribute' section of a class definition. The outline for defining an attribute is given below.

```
ATTRIBUTE_NAME[ // the name of the attribute
  _atr_type STRING  // the type of the attribute - must be given first
  _is_optional BOOLEAN // indicates if the attribute is optional or mandatory
  _constraints [ ... ] // constraints on the value of the attribute
  _attributes [ ... ] // the inner attributes for a composite attribute
]
```

To define an attribute, the name of the attribute is given as a key. The name of the attribute must match the 'ATTRIBUTE_NAME' production of the ANML syntax given in Figure 9. As the value, a sequence of key-value pairs is given indicating the type of the attribute and any constraints. The type must be given first using the reserved key '_atr_type'. An attribute can be one of several types which are listed in Section 5.9. The remaining specifications are optional and may be given in any order.

The '_is_optional' reserved key is used to indicate if an attribute is optional or mandatory. Indicating that an attribute is optional means that a value for the given attribute need not be assigned. By default, all attributes are mandatory so this specification only needs to be given if the attribute is optional. To do this 'true' must be given as the boolean value.

If the attribute is mandatory then the attribute must be assigned in a component instance for the instance to be valid. A mandatory attribute does not necessarily have to be assigned by the user though. If the mandatory attribute has been assigned a default value in the '_default' section of a '_class' definition, the attribute will automatically be assigned the default value in a component instance if the user does not assign another value.

Various constraints can be specified to restrict the allowable values of an attribute by using the reserved key '_constraints'. A description of the different kinds of attribute constraints can be found in Section 5.8. In the case that an attribute is a composite attribute, which is specified by the type 'comp_atr', the '_constraints' reserved key cannot be used. A composite attribute is an attribute that has internal attributes. Instead of using the '_constraints' reserved key, the '_attributes' reserved key is used to define the attributes internal to the attribute. Internal attributes may have constraints imposed as long as they are not composite attributes.

There are no attributes that we wish to associate with the ‘Network’ class so we do not define an ‘_attributes’ section. On the other hand, we wish to associate some attributes with the ‘Router’ class and define them below.

```
_attributes [

  /* the time it takes to process a packet in the router in seconds. */
  proc_delay [
    _atr_type real
    _constraints [ _min_exclusive 0 ]
  ]

  /* size of buffer on each router interface in Bytes. */
  buffer_size [
    _atr_type real
    _constraints [ _min_exclusive 0 ]
  ]

  /* the lan_links a router is attached to */
  lan_links [
    _atr_type id_type
    _is_optional true
    _constraints [ _valid_classes {LAN_Link} ]
  ]

] // end Router _attributes
```

We want the ‘proc_delay’ and ‘buffer_size’ attributes to be mandatory so we omit the ‘_is_optional’ key. Both of these attributes are of the ‘real’ (real number) type and we want their values to be strictly greater than zero. The ‘_min_exclusive’ constraint is used to specify this. We want the ‘lan_links’ attribute to be optional as a router might not be connected to a LAN so we specify the ‘_is_optional’ key as being ‘true’. The type of the ‘lan_links’ attribute is the ‘id_type’ which means that values assigned to this attribute are identifiers that point to other components in the model. To ensure that only components of type ‘LAN_Link’ may be pointed to the ‘_valid_class’ constraint is used.

2.2.1.3 Assigning a Default Attribute Value

Default attributes are assigned in the ‘_default’ section of a class definition. Only attributes which are not of the ‘id_type’ can be given default values since identifiers are only known when a model is being created. Inherited attributes may also be assigned default values. If the inherited attribute was assigned a default value in an ancestor class, it is overridden by the newly assigned value for this class and its descendants.

Default values should only be assigned to mandatory attributes. If a default value is assigned to an optional attribute the attribute is no longer optional as a value will always be given to that attribute even if the user does not assign one.

The ‘Network’ class has no attributes associated with it so the ‘_default’ section is not defined. The ‘Router’ class does have attributes associated with it and the ‘_default’ section is defined as below.

```
_default[

  proc_delay 0.000001  // seconds
  buffer_size 51200   // 51200 Bytes = 50 kB

] // end Router _default
```

Default values are assigned to the ‘proc_delay’ and ‘buffer_size’ attributes of the ‘Router’ class. Therefore if either of these attributes are not assigned in a ‘Router’ definition or instance, the default values indicated above will automatically be assigned.

2.2.2 Declaring a Valid Top Level Component

The last thing to do when defining a schema is to declare the components that are allowed at the top level of a model. Top level components refer to those components that are not instantiated in another component, or are at the top of the component hierarchy. This is done in the same manner as declaration of components in a class definition. (See Section 2.2.1.1.) The top level component declarations for the NetSchema are given below.

```
_components [
  /* Allow only a single Network instance at top level
   */
  Network [ _occurs [ _num_occur 1] ]

] // end _schema _components
```

We want to model networks so we only allow 'Network' instances at the top level. The '_num_occur' constraint is used to indicate that only one instance of 'Network' is allowed. This network of course can be composed of many sub-networks.

3 Tutorial: How to Create an ANML Model

This tutorial describes a suggested process to follow when creating an ANML model. Before creating an ANML model one should be familiar with the ANML Schema that has been defined for the model being created. It is the ANML Schema that specifies what components and attributes may exist in a model. (See Section 2.2).

An example simulation scenario based on the network schema used in the schema tutorial is used to demonstrate this process. The network schema definition file and the full versions of the files used to create the example network can be found in appendix A.

This section proceeds by explaining the four main steps used to create an ANML model. The first step is deciding what to model. The second step is breaking the model into individual components. The third step is defining database components needed for the model. Finally, the fourth step is defining the actual model.

3.1 Step 1 - Decide What to Model

The first step in creating an ANML model is deciding what to model. For example, if one is planning to create a network model, the topology of the network has to be decided upon. In this case drawing a digram of the network topology is the easiest way to do this. The topology of the example network used for this tutorial is given in Figure 2.

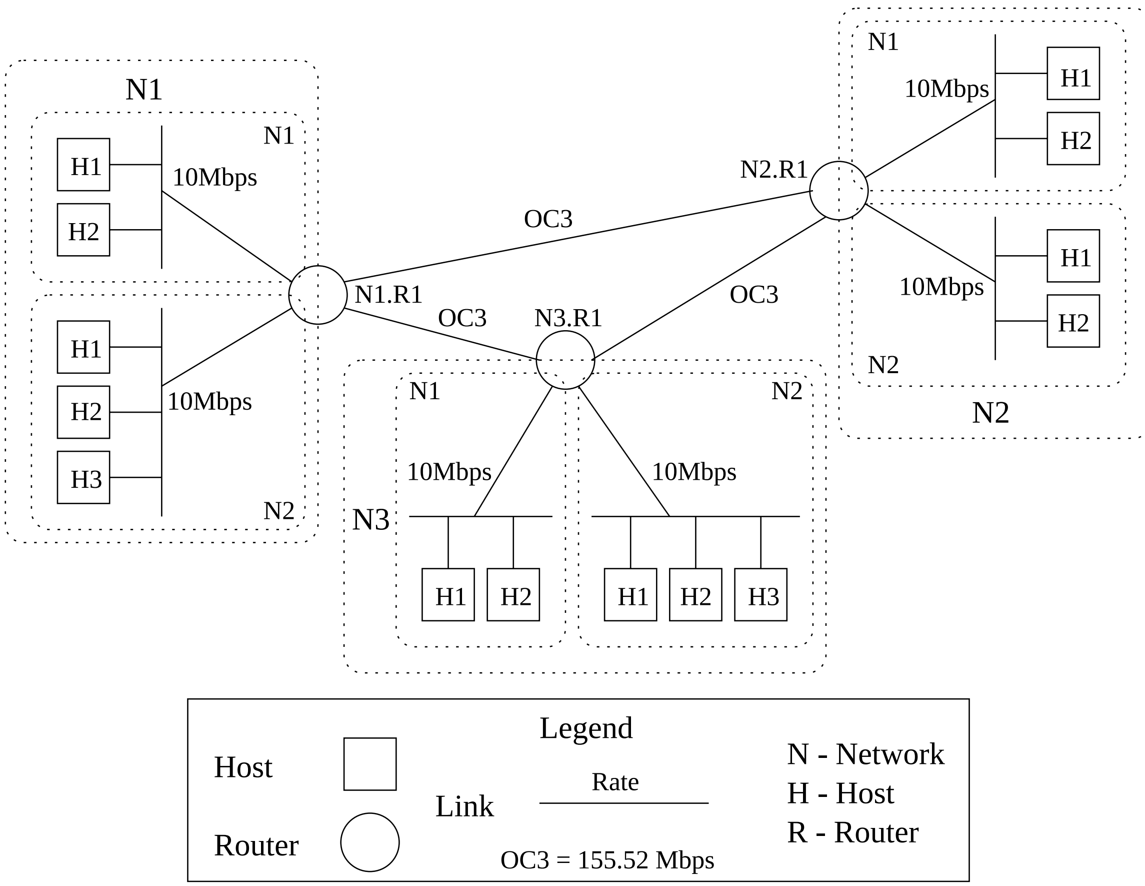

Figure 2: An Example Network Topology

3.2 Step 2 - Break the Model Down into Individual Components

After deciding what to model, the model needs to be broken into individual components. Start by identifying the individual top level components. Then for each individual top level component, identify the individual sub-components. Continue this process until components that have no sub-components are reached. Special attention should be given for finding components that are identical.

For a network model this involves breaking the network into its subnets, and breaking the subnets into their subnets, and so on. Care must be taken to look for subnets that are identical. Also, the different types of nodes and links on the network must be examined.

By examining the network topology given in Figure 2 we can see that there are three main networks, which are further broken into subnets. There are subnets that have two hosts and subnets that have three hosts. Notice that network 1 and network 3 are identical. Three routers form a backbone that joins the three networks together. We can also see that there are two different types of links. There are OC-3 point-to-point links that join the routers together and 10Mbps links that join the nodes of the small subnets or local area networks (LANs) together. Figure 3 shows how the example network has been decomposed.

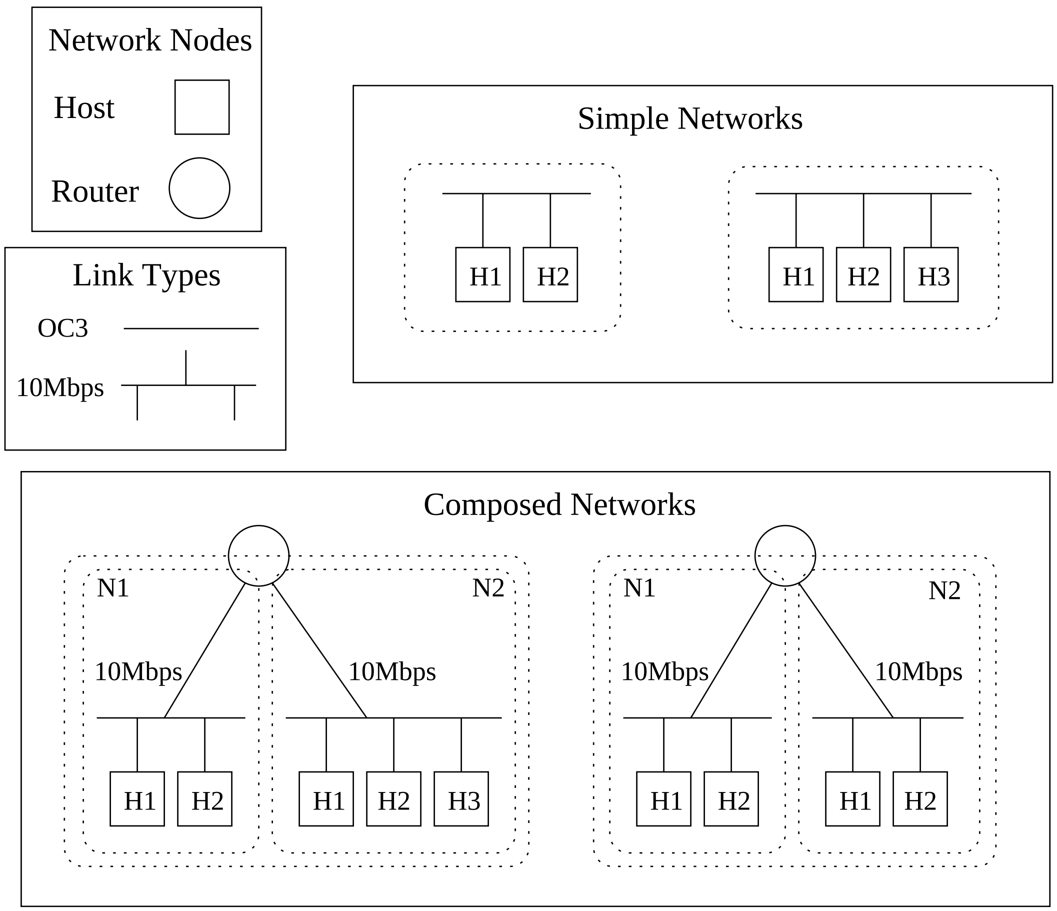

Figure 3: Decomposition of Example Network Topology

3.3 Step 3 - Define Components Needed by the Model in a Database

After breaking a model into individual components, these components can be defined in database structures. A database structure exists to serve as a repository of component definitions for use in constructing models. This allows for components to be defined only once and used repeatedly. Databases can be kept for future use to allow for easy construction of other models. Note that it is not necessary for components to be defined in databases. Components can be instantiated based on the schema alone. Figure 4 illustrates model dependencies. Models can be created based on the schema alone, based on both the schema and databases, or based on databases alone. Databases always depend on the schema and possibly on other databases as well.

An outline for defining a database is given below.

```
_database [
```

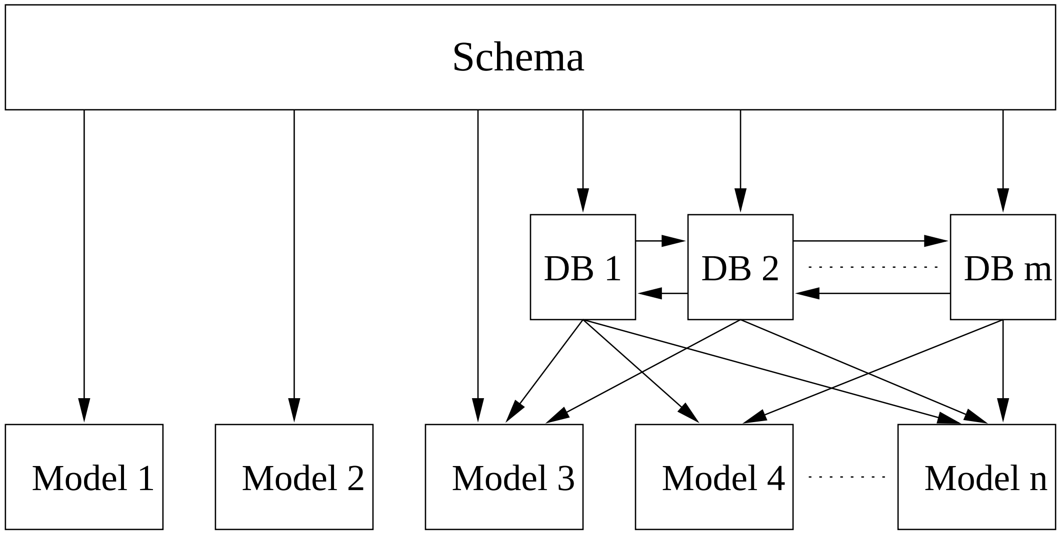

Figure 4: Model Dependencies

```
  _name STRING // the name of the database - must be the first entry
  DatabaseEntry1 []
  DatabaseEntry2 []
  ...
]
```

Databases are defined using the reserved key '_database'. The value associated with the '_database' key is a key-value list. The first entry of the key-value list is the '_name' reserved key which assigns a name to the database. No two databases may have the same name. The remaining entries of the key-value list are the component definitions. An outline for a component definition is given below.

```
COMPONENT_NAME [
  _class STRING // the class the component belongs to - must be the first entry

  ... // sub-component instances and attribute assignments
]
```

A component is defined by first giving it a name as the key. Each component definition within a given database must have a unique name. Also, the name may not be the same as that of a class defined in the schema being used. For the value, a key-value list is given of which the first key-value pair must specify the class this component belongs to. This is done using the reserved key '_class'. The class must be a class that has been defined in the schema that is being used. The remaining key-value pairs specify the desired sub-components and attributes for the component definition. Only sub-components and attributes which are specified in this component's class definition in the schema may be used.

Indivisible components of the model are defined first. These components are then used to define the larger components. Nodes and links are indivisible components, so let us first define a database for the different types of nodes on the network.

```
/*-------------------------------------------------------------------------------
 * Database: Node_DB
 *
 * Description: Contains definitions of different network nodes.
 *-------------------------------------------------------------------------------
 */
_database [
  _name Node_DB
```

```
  StdHost[ _class Host buffer_size 102400 ]

  StdRouter[ _class Router proc_delay 0.000005 buffer_size 102400 ]

] // end _database Node_DB
```

One type of host and one type of router have been defined for use in the model. These components have no sub-components but do have some associated attributes. The host has a buffer size associated with it while the router has a processing delay and a buffer size associated with it. Attributes are assigned by giving the name of the attribute and then the value. The allowable value depends on the type of the attribute and the constraints imposed on the attribute as specified in the schema definition. Further information pertaining to attribute constraints and types can be found in Sections 5.8 and 5.9.

Let us now create the database for the other group of indivisible components, namely, the links.

```
/*-----------------------------------------------------------------------
 * Database: Link_DB
 *
 * Description: Contains definitions of different links.
 *-----------------------------------------------------------------------
 */
_database [
  _name Link_DB

  /* 10Mbps LAN_Link (representative of 10Mbps Ethernet) */
  LAN_10Mbps[ _class LAN_Link rate 10]

  /* an OC-3 point-to-point link - 155.52 Mbps */
  Link_OC3[ _class P2P_Link rate 155.52 ]

] // end _database Link_DB
```

A 10Mbps LAN link and an OC-3 point-to-point link are defined to represent the two types of links in the network model. By examining the class definitions of 'LAN_Link' and 'P2P_Link' in appendix A we can see that default values for the 'mtu' attribute of 1500 and 9180 respectively have been given. As the link definitions in the database do not assign the 'mtu' attribute, the default values are automatically assigned. Attributes only need to be assigned if they have a value different from the default value.

Now that we have databases for the different network nodes and links, we can start to define and compose the different networks. Let us start with the simple networks as labeled in Figure 3. First of all, we will define the network that has only two hosts as shown in Figure 5. Let us name it 'Net2H' to represent that it has two hosts.

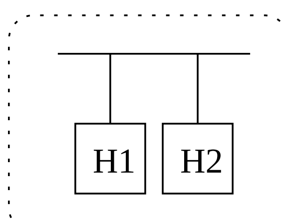

Figure 5: Simple Network with 2 Hosts

```
/*-----------------------------------------------------------------------
 * Database: Network_DB
 *
```

```
 * Description: Contains definitions of various networks.
 *-----------------------------------------------------------------------------
 */
_database [
  _name Network_DB

  /* A LAN with two hosts */
  Net2H[
    _class Network
    LAN_10Mbps[ _id L1 _in_database Link_DB delay 0.00001]
    StdHost[ _id {H1,H2} _in_database Node_DB lan_link .L1]
  ]

  ...
```

The 'Net2H' component definition demonstrates the instantiation of sub-components in a component definition. Components may be instantiated with or without the use of a database component definition. All of the sub-component instances in 'Net2H' were created using database component definitions. When instantiating a component using a component definition all of the attributes and sub-components specified in the component definition are adopted by the instance. In essence, a copy of the definition is made to which additional sub-components and attributes may be added. A general outline for creating component instances using component definitions is given below.

```
COMPONENT_NAME [ // the name of the component definition being used
  _id value // the id of the component instance - must be the first entry
  _in_database STRING // the name of database component definition is in -
                      // must be the second entry

  ... // additional sub-component instances and attribute assignments

]
```

When instantiating a component using a component definition, the name of the component definition is given as the key. As the value, a key-value list is given in which the first entry specifies the identifier of the component and the second entry specifies the database that this component definition is contained in. The identifier of the component must be unique to the level that it is on. The remaining entries of the list are additional sub-component instances and attribute assignments above what is already contained in the component definition. These additional sub-components and attributes must be specified in the component's class definition in the schema in order to be used. It is not necessary to specify additional sub-components and attributes though. Both the 'LAN_10Mbps' and 'StdHost' instances in 'Net2H' add an extra attribute assignment. The 'LAN_10Mbps' instance assigns a delay to the link. The 'StdHost' instance specifies the LAN link that it is connected to.

It is also possible to create a component instance without using a component definition (i.e., dependent on the schema only). The general outline for creating a component instance in this manner is given below.

```
COMPONENT_NAME [ // the name of the class component instance belongs to
  _id value // the id of the component instance - must be the first entry

  ... // sub-component instances and attribute assignments

]
```

When instantiating a component without using a component definition, the name of the class the component instance belongs to is given as the key. As the value, a key-value list is given in which the first entry specifies the identifier of the component. The remaining key-value pairs specify the desired sub-components and attributes for the component instance. Only sub-components and attributes which are specified in this component's class definition in the schema may be used.

For example, the 'LAN_10Mbps' sub-component instance of 'Net2H' could have been instantiated to create the exact same instance without using a database definition as follows.

```
LAN_Link [ _id L1 rate 10 delay 0.00001 ]
```

The advantages of using database component definitions, are that they can be used over and over, the attributes and sub-components do not need to be given every time and they help maintain consistency in a model. Defining components in databases is especially helpful when components are large and complicated.

The definition of 'Net2H' also represents two different ways of specifying the '_id' of a component. Specifying a single identifier creates a single component instance. For example, 'L1' was specified as an identifier for 'LAN_10Mbps' so only one instance of 'LAN_10Mbps' is created with identifier 'L1'. On the other hand, specifying a list of identifiers creates an instance for every identifier in the list. For example, 'StdHost' has a list of identifiers specified ('{H1,H2}') and thus two instances of 'StdHost' are created with identifiers 'H1' and 'H2'. This is a tool to help easily create multiple instances of a component.

Another thing to note is the use of an identifier in the 'lan_link' attribute of the 'StdHost' instances.

```
StdHost[ _id {H1,H2} _in_database Node_DB lan_link .L1]
```

The 'lan_link' attribute is of the type 'id_type'. This means that the value of the attribute is an identifier of another component in the model. When an 'id_type' attribute begins with a '.' it refers to a component instance beginning at the same level of the component instance the attribute is assigned in. So in the above example, the '.L1' identifier refers to a component instance at the same level as the 'StdHost' instances. Thus the '.L1' is referring to the 'LAN_10Mbps' instance with identifier 'L1' that is instantiated at the same level as the 'LAN_10Mbps' instances. If a '.' does not precede an 'id_type' attribute value, then it refers to a component instance beginning at the sub-component level of the component instance the attribute is in. More information on the 'id_type' can be found in Section 5.9.1.5.

Continuing on, let us define the network with three hosts as shown in Figure 6. Let us give it the name 'Net3H' to represent that it has three hosts.

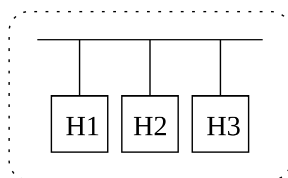

Figure 6: Simple Network with 3 Hosts

```
/* A LAN with three hosts */
Net3H[
  _class Network
  LAN_10Mbps[ _id L1 _in_database Link_DB delay 0.00001]
  StdHost[ _id [_from H1 _to H3] _in_database Node_DB lan_link .L1]
]
```

The above example demonstrates the third and last way that the identifier of a component may be specified. Besides specifying a single identifier or a list of identifiers, a range of identifiers can be specified. By specifying a range of identifiers a component for each identifier in the range is created. In the above example three instances of 'StdHost' are created with identifiers 'H1', 'H2' and 'H3'.

Now that the simple networks have been defined, we can define the larger networks which are composed of the smaller networks. These networks can be seen in Figure 3 labeled as 'Composed Networks'. Let us start by defining the network that has the two identical subnets as shown in Figure 7. We will name it 'Net2S_A' to represent that it has two subnets and to distinguish it from the other network type with two subnets. Defining this network involves using the simple network 'Net2H' that was already defined.

```
/* A network with two subnets, both having two hosts each. A router
 * joins the two subnets.
 */
Net2S_A[
```

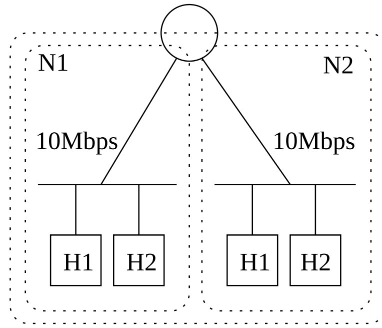

Figure 7: Network Composed of Two Identical Subnets

```
    _class Network
    Net2H[ _id {N1,N2} _in_database Network_DB]
    StdRouter[ _id R1 _in_database Node_DB lan_links{.N1.L1, .N2.L1} ]
  ]
```

The 'lan_links' attribute of the 'StdRouter' instance is of the 'id_type' and has a list of identifiers as its value. Just as in specifying the identifier of a component, the value of an 'id_type' attribute may be a single identifier, a list of identifiers or a range of identifiers. The identifiers referred to by the 'lan_links' attribute also demonstrate the use of hierarchical identifiers, or an identifier that has more than one level. A '.' separates the levels of an identifier. Let us examine the identifier '.N1.L1'. The '.N1' refers to the 'Net2H' instance with identifier 'N1'. The 'L1' then refers to the 'LAN_10Mbps' sub-component instance of 'Net2H' with identifier 'L1'. More information on hierarchical identifiers can be found in Section 4.7.

Last of all, let us define the network that has the two differing subnets as shown in Figure 8. Let us name it 'Net2S_B' to represent that it has two subnets, and to distinguish it from the other network type with two subnets.

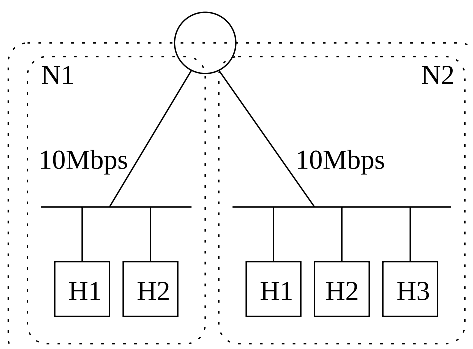

Figure 8: Network Composed of Two Different Subnets

```
  /* A network with two subnets, one with two hosts, and one with three hosts.
   * A router joins the two subnets.
   */
  Net2S_B[
    _class Network
    Net2H[ _id N1 _in_database Network_DB]
    Net3H[ _id N2 _in_database Network_DB]
    StdRouter[ _id R1 _in_database Node_DB lan_links{.N1.L1, .N2.L1} ]
  ]

] // end _database Network_DB
```

3.4 Step 4 - Define the Model

After all the necessary database components have been defined, the model can now be defined. As database components are designed for easy reuse, the model should be defined in a separate file from the database file or files. The model file can include the database files that it needs. The model file should also include the file which has the ANML schema definition to be used. Inclusion of files is done as demonstrated below.

```
_include "net_schema.anml"
_include "net_databases.anml"
```

A file can be included using the '_include' reserved keyword. The name of the file to be included is then specified as the value.

An outline for defining a model is given below.

```
_model [
  _name STRING // the name of the model
  _use_schema STRING // the name of the schema used to validate the model

  ... // component instances in model
]
```

Models are defined using the reserved key '_model'. The value associated with the '_model' key is a key-value list. A name is assigned to the model using the reserved key '_name' and the name of the schema that will be used to validate the model is given using the '_use_schema' reserved key. The network model given in Figure 2 is defined below. The definition of the model is quick and concise as it simply uses the components that were previously defined in the databases.

```
_model[
  _name tut1_model
  _use_schema NetSchema

  Network[
    _id N

    /* The model is comprised of three main networks */
    Net2S_B[ _id {N1,N3} _in_database Network_DB ]
    Net2S_A[ _id N2 _in_database Network_DB ]

    /* The links to join the three networks together */
    Link_OC3[ _id L1 _in_database Link_DB delay 0.0001 nodeA .N1.R1 nodeB .N3.R1]
    Link_OC3[ _id L2 _in_database Link_DB delay 0.0002 nodeA .N1.R1 nodeB .N2.R1]
    Link_OC3[ _id L3 _in_database Link_DB delay 0.00015 nodeA .N2.R1 nodeB .N3.R1]
  ]
]
```

The model that we were trying to create has now been completed. We started by defining the smaller components and then used them as building blocks in creating the model. Also, should we wish to create other models in the future, we already have some database components defined for easy reuse. Proper utilization of the database can be of great help in constructing large and complex models.

4 A Formal Definition of ANML

The syntax for ANML is primarily based on DML (Domain Modeling Language) [7] with some minor modifications. Both languages follow a key-value pair concept, but ANML adds a further classification of key names, and also adds the capability of having a list of values. ANML also uses ideas from XML (Extensible Markup Language) [9]. In XML, documents are composed of elements which have associated attributes. Elements can further be composed of other elements. Similarly, an ANML model is composed of components which have associated attributes. Components can further be composed of other components. This section proceeds by introducing the syntax for ANML and then by examining each production of the syntax in detail.

4.1 ANML Syntax

The syntax for ANML using Extended Backus-Naur Form (EBNF) notation can be seen in Figure 9.

```
           model  ::=  key_value_list

  key_value_list  ::=  key_value_pair key_value_list
                   |   key_value_pair

  key_value_pair  ::=  key value

             key  ::=  COMPONENT_NAME
                   |   ATTRIBUTE_NAME
                   |   RESERVED_NAME

           value  ::=  STRING
                   |   “[” key_value_list “]”
                   |   “[” “]”
                   |   “{” value_list “}”
                   |   “{” “}”

      value_list  ::=  value_list ”,” value
                   |   value

  COMPONENT_NAME  ::=  [A-Z][-_a-zA-Z0-9]*

  ATTRIBUTE_NAME  ::=  [a-z][-_a-zA-Z0-9]*

   RESERVED_NAME  ::=  _[-_a-zA-Z0-9]*

          STRING  ::=  \“[ˆ\t\n\”]*\”
                   |   [ˆ \t\n\[\]\{\}\/\”,#]*
```

Figure 9: ANML Syntax

An ANML file, or set of files, defines a model through a list of key-value pairs. A key is a string of symbols that is used to identify components of a model, attributes associated with a component, or other important information. Associated with every key is a value that provides information pertaining to the key. Models are hierarchically defined by allowing components to consist of sub-components.

4.2 Well-Formedness and Validity

An ANML model is considered to be well-formed if it matches the ‘model’ production of the syntax given in Figure 9 and if it satisfies any additional well-formedness constraints. A well-formed model is also referred to as a syntactically correct model.

An ANML model is considered to be valid if it satisfies validity constraints and any other constraints imposed by the ANML schema definition. A valid model is also referred to as a schematically correct model.

4.3 Model and Key-Value Pairs

The top level non-terminal of the ANML syntax is ‘model’. A model is what one wishes to define using ANML, such as a network topology. Models are described via a list of key-value pairs which are defined in an ANML file or set of ANML files.

Model		
model	::=	key_value_list

Validity Constraint: All keys appearing in the top level of the ‘model’ key-value list must either be component keys or reserved keys. Attributes may not be assigned at the top level of a model.

A key-value pair list must consist of at least one key-value pair, but is not limited in size. In general, some form of white space, such as a space, tab or newline, is required between key-value pairs to distinguish them.

Key-Value Pair List		
key_value_list	::=	key_value_pair key_value_list
	|	key_value_pair

A key-value pair consists of a key followed by a value. In general, some form of white space, such as a space, tab or newline, is required between a key and a value to distinguish them.

Key-Value Pair		
key_value_pair	::=	key value

4.4 Keys

A key is a string of symbols that is used to identify a certain piece of information. There are three types of keys: component keys, attribute keys and reserved keys. A component key is the name of a component in the model such as a ‘Network’ or ‘Host’. An attribute key is the name of an attribute associated with a component, such as the ‘buffer_size’ of a ‘Host’. The reserved keys are predefined keys that specify certain information for use by the ANML Processor. Each of the key types begins with a different type of character so as to more easily distinguish them.

Key		
key	::=	COMPONENT_NAME
	|	ATTRIBUTE_NAME
	|	RESERVED_NAME

4.4.1 Component Keys

All component keys must begin with an upper case letter and may be followed by any pattern of alpha-numeric characters, underscores (‘_’) or hyphens (‘-’). A component key represents the name of a component in the model such as ‘Network’ or ‘Host’. The value associated with a component key describes the component by specifying any attributes associated with the component, and also by specifying any sub-components.

Component Name Key		
COMPONENT_NAME	::=	[A-Z][-_a-zA-Z0-9]*

Components are referred to in one of two ways: a component definition or a component instance. A component definition may be created within a database to define a specific component with its associated attributes and sub-components. A component instance is a component that occurs within a model and must be given a unique identifier. Component definitions may be used in creating component instances.

4.4.1.1 Component Definitions

Validity Constraint: A component definition must match the following syntax:

Component Definition Syntax		
comp_definition	::=	COMPONENT_NAME "[" _CLASS_KEY STRING ... "]"

where '...' refers to a continuation of the key-value pair list. Zero or more additional key-value pairs may occur in the list.

The 'COMPONENT_NAME' specifies the name of the component definition and the 'STRING' value associated with the '_CLASS_KEY' specifies the class that the component belongs to. The desired sub-components and attributes are then entered as the remaining key-value pairs.

Validity Constraint: A component definition may only occur as a top level entry of a database definition.

Validity Constraint: The 'COMPONENT_NAME' of a component definition must not be the same as the name of a class specified in the schema definition for the model. The 'COMPONENT_NAME' must also not be the same name as any other component defined in the same database.

4.4.1.2 Component Instances

Validity Constraint: A component instance must follow the following syntax:

Component Instance Syntax		
comp_instance	::=	COMPONENT_NAME "[" _ID_KEY value ... "]"
	|	COMPONENT_NAME "[" _ID_KEY value _IN_DATABASE_KEY STRING ... "]"

where '...' refers to a continuation of the key-value pair list. Zero or more additional key-value pairs may occur in the list.

A component may be instantiated with or without using component definition. If instantiating a component without using a component definition, the first production above is used. The 'COMPONENT_NAME' is the name of the class the component instance belongs to. The value for the '_ID_KEY' is the identifier of the component for the hierarchical level that it is defined in. The desired sub-components and attributes for the component instance are then entered as the remaining key-value pairs.

If instantiating a component using a component definition, the second production above is used. The 'COMPONENT_NAME' is the name of the component definition that is being used to create the component instance. The value for the '_ID_KEY' is the identifier of the component for the hierarchical level that it is defined in. The name of the database that the component is defined in is given by the 'STRING' value of the '_IN_DATABASE_KEY'. Any desired additional sub-components and attributes, or modifications to the component definition being used, are then entered as the remaining key-value pairs.

Validity Constraint: A component instance may not occur at a top level of a database.

Validity Constraint: If the '_IN_DATABASE_KEY' is not specified the 'COMPONENT_NAME' must be the name of a class specified within the ANML schema definition for the model.

Validity Constraint: If the '_IN_DATABASE_KEY' is specified within the component instance then the 'COMPONENT_NAME' must be the name of a component definition within the specified database.

Validity Constraint: The identifiers specified in the value associated with the '_ID_KEY' of a component instance must be unique. If the component is a sub-component, then it's identifier must be different from the identifiers of all other sub-components of the same component. If the component is a top level component in the model, then it's identifier must be different from the identifiers of all other top level components in the model.

4.4.2 Attribute Keys

All attribute keys must begin with a lower case letter and may be followed by any pattern of alpha-numeric characters, underscores ('_') or hyphens ('-'). An attribute describes a characteristic of a certain component. The 'buffer_size' of a 'Host' is an example of an attribute. If an attribute is assigned twice within the same component, the old assignment of the attribute is overridden.

Attribute Name Key		
ATTRIBUTE_NAME	::=	[a-z][-_a-zA-Z0-9]*

4.4.3 Reserved Keys

All reserved keys must begin with an underscore ('_') and may be followed by any pattern of alpha-numeric characters, underscores ('_') or hyphens ('-'). Reserved keys are keys which are specific to the ANML Processor. They are used to specify various information needed by the processor, such as the schema definition, identifiers, and inclusion of other files.

Reserved Name Key		
RESERVED_NAME	::=	_[-_a-zA-Z0-9]*

The reserved keys, excluding those used in the ANML Schema definition are given below.

4.4.3.1 _CLASS_KEY

The '_CLASS_KEY' is used in a component definition to specify the class that the component belongs to.

Class Key		
_CLASS_KEY	::=	"_class"

Validity Constraint: The 'class' key-value pair must match the following syntax:

Class Key-Value Syntax		
class_key_value	::=	_CLASS_KEY STRING

The 'STRING' value of the '_CLASS_KEY' is the name of the class that the component belongs to.

Validity Constraint: The 'STRING' value associated with the '_CLASS_KEY' of a component definition must be the name of a class specified within the schema definition for the model.

4.4.3.2 _DATABASE_KEY

The '_DATABASE_KEY' is used to define a database. Within the database component definitions can be created for easy re-use and creation of complex components.

Database Key		
_DATABASE_KEY	::=	"_database"

Validity Constraint: The 'database' key-value pair must match the following syntax:

Database Key-Value Syntax		
database_key_value	::=	_DATABASE_KEY "[" _NAME_KEY STRING ... "]"

where '...' refers to a continuation of the key-value pair list. Zero or more additional key-value pairs may occur in the list. These additional key-value pairs are the component definitions within the database. (See Section 4.4.1.1.)

The 'STRING' value associated with the '_NAME_KEY' is the name of the database.

Validity Constraint: The 'STRING' value associated with the '_NAME' key must be unique among all of the database names.

4.4.3.3 _FROM_KEY

The ‘_FROM_KEY’ is used when specifying an identifier range, and indicates the the start of the range.

From Key		
_FROM_KEY	::=	“_from”

Validity Constraint: The ‘from’ key-value pair must match the following syntax:

From Key-Value Syntax		
from_key_value	::=	_FROM_KEY HIERARCHICAL_ID

The ‘HIERARCHICAL_ID’ value associated with the ‘_FROM_KEY’ is the identifier of the component to begin the range from. Note that the value of the ‘_FROM_KEY’ must be a ‘LEVEL_ID’ when used with the ‘_ID_KEY’ in specifying the identifier of a component instance (see Section 4.4.3.4). For information on ‘HIERARCHICAL_ID’s and ‘LEVEL_ID’s see Section 4.7.

Validity Constraint: The ‘HIERARCHICAL_ID’ associated with the ‘_FROM_KEY’ must consist of two parts. The prefix of the identifier must consist of a string of characters not ending with a digit. The suffix of the identifier must be a number.

4.4.3.4 _ID_KEY

The ‘_ID_KEY’ is used to specify the identifier of a component instance. A single identifier, a list of identifiers or a range or identifiers may be specified to ease the process of creating multiple instances.

Identifier Key		
_ID_KEY	::=	“_id”

Validity Constraint: The identifier key-value pair must match the following syntax:

Identifier Key-Value Syntax		
id_key_value	::=	_ID_KEY LEVEL_ID
	|	_ID_KEY “{” LEVEL_ID, ... “}”
	|	_ID_KEY “[” _FROM_KEY LEVEL_ID _TO_KEY LEVEL_ID “]”

where ‘...’ refers to a continuation of the ‘LEVEL_ID’ value list. All values in the list must be ‘LEVEL_ID’s. The list may be empty. For information on ‘LEVEL_ID’s see Section 4.7.

When creating a single instance of a component the first production above is used. The ‘LEVEL_ID’ value of the ‘_ID_KEY’ is the identifier that is being given to the component instance.

When creating multiple instances of a component where the identifiers are not contiguous, the second production above is used. A component for each ‘LEVEL_ID’ value in the list is instantiated.

When creating multiple instances of a component where the identifiers are contiguous the third production above is used. The ‘LEVEL_ID’ value of the ‘_FROM_KEY’ specifies the first identifier of the range. The ‘LEVEL_ID’ value of the ‘_TO_KEY’ specifies the last identifier of the range. A component for each identifier in the range is instantiated.

Validity Constraint The string prefixes of the ‘LEVEL_ID’ values associated with the ‘_FROM_KEY’ and ‘_TO_KEY’ must be the same.

Validity Constraint The number suffix of the ‘LEVEL_ID’ value for the ‘_TO_KEY’ must be greater than or equal to the number suffix of the ‘LEVEL_ID’ value for the ‘_FROM_KEY’.

4.4.3.5 _IN_DATABASE_KEY

The ‘_IN_DATABASE_KEY’ is used when using a component definition to create a component instance. The value of the ‘_IN_DATABASE_KEY’ specifies the name of the database that the component definition being used is in.

In Database Key		
_IN_DATABASE_KEY	::=	"_in_database"

Validity Constraint: The 'in database' key-value pair must match the following syntax:

In Database Key-Value Syntax		
in_database_key_value	::=	_IN_DATABASE_KEY STRING

The 'STRING' value associated with the '_IN_DATABASE_KEY' is the name of the database that the component definition being used is in.

Validity Constraint: The 'STRING' value associated with the '_IN_DATABASE_KEY' must be the name of a database that has been defined.

4.4.3.6 _INCLUDE_KEY

The '_INCLUDE_KEY' is used to include other files that contain definitions that are being used in the current file.

Include Key		
_INCLUDE_KEY	::=	"_include"

Well-Formedness Constraint: The 'include' key-value pair must match the following syntax:

Include Key-Value Syntax		
include_key_value	::=	_INCLUDE_KEY STRING

The 'STRING' value associated with the '_INCLUDE_KEY' is the name of the file to include.

Well-Formedness Constraint: The 'STRING' value associated with the '_INCLUDE_KEY' must be the name of an existing file that has the extension '.anml'. The relative or absolute path to the file must be included as part of the file name.

4.4.3.7 _MODEL_KEY

The '_MODEL_KEY' is used to define a model.

Model Key		
_MODEL_KEY	::=	"_model"

Validity Constraint: The 'model' key-value pair must match the following syntax:

Model Key-Value Syntax		
model_key_value	::=	_MODEL_KEY "[" _NAME_KEY STRING _USE_SCHEMA_KEY STRING ... "]"

where '...' refers to a continuation of the key-value pair list which represents the components of the model. The 'STRING' value associated with the '_NAME_KEY' is the name of the model and the 'STRING' value associated with the '_USE_SCHEMA_KEY' is the name of the schema that will be used to modify the model.

4.4.3.8 _NAME_KEY

The '_NAME_KEY' is used in specifying the name of a model, database or schema.

Name Key		
_NAME_KEY	::=	"_name"

Validity Constraint: The 'name' key-value pair must match the following syntax:

Name Key-Value Syntax		
name_key_value	::=	_NAME_KEY STRING

The 'STRING' value associated with the '_NAME_KEY' is the name of the model, database or schema.

4.4.3.9 _TO_KEY

The ‘_TO_KEY’ is used to specify the upper end of an identifier range.

To Key		
_TO_KEY	::=	“_to”

Validity Constraint: The ‘to’ key-value pair must match the following syntax:

To Key-Value Syntax		
to_key_value	::=	_TO_KEY HIERARCHICAL_ID

The ‘HIERARCHICAL_ID’ associated with the ‘_TO_KEY’ specifies the identifier of the component of the upper end of the identifier range. Note that the value of the ‘_TO_KEY’ must be a ‘LEVEL_ID’ when used with the ‘_ID_KEY’ in specifying the identifier of a component instance (See Section 4.4.3.4). For information on ‘HIERARCHICAL_ID’s and ‘LEVEL_ID’s see Section 4.7.

Validity Constraint: The ‘HIERARCHICAL_ID’ associated with the ‘_TO_KEY’ must consist of two parts. The prefix of the identifier must consist of a string of characters not ending with a digit. The suffix of the identifier must be a number.

4.4.3.10 _USE_SCHEMA_KEY

The ‘_USE_SCHEMA_KEY’ is used in a model definition to specify the name of the schema to use to validate the model.

Use Schema Key		
_USE_SCHEMA_KEY	::=	“_use_schema”

Validity Constraint: The ‘use schema’ key-value pair must match the following syntax:

Use Schema Key-Value Syntax		
use_schema_key_value	::=	_USE_SCHEMA_KEY STRING

The ‘STRING’ value associated with the ‘_USE_SCHEMA_KEY’ is the name of the schema that will be used to validate the model.

Validity Constraint: The ‘STRING’ value associated with the ‘_USE_SCHEMA_KEY’ must be the name of a schema that has been defined.

4.5 Values

Values are always associated with a key and represent information pertaining to that key. There are three main types of values: string values, key-value pair lists and value lists. Empty key-value pair lists or value lists are also acceptable as values.

<table>
<tr><th colspan="3">Value</th></tr>
<tr><td>value</td><td>::=</td><td>STRING</td></tr>
<tr><td></td><td>|</td><td>“[” key_value_list “]”</td></tr>
<tr><td></td><td>|</td><td>“[” “]”</td></tr>
<tr><td></td><td>|</td><td>“{” value_list “}”</td></tr>
<tr><td></td><td>|</td><td>“{” “}”</td></tr>
</table>

A value list is a list of values. Each value in the list is separated by a comma.

<table>
<tr><th colspan="3">Value List</th></tr>
<tr><td>value_list</td><td>::=</td><td>value_list “,” value</td></tr>
<tr><td></td><td>|</td><td>value</td></tr>
</table>

Validity Constraint: All values in a value list must be 'STRING's for a model to be valid. However, a model is still well-formed if key-value pair lists and value lists are used as values in a value list. The syntax maintains support of all types of values in a value list to easily allow for future extension if needed.

The terminal value is a string. If white space, square brackets, curly brackets, commas, slashes ('/') or hashes ('#') appear in the string, the string needs to be put inside of quotes, otherwise quotes are not necessary.

<table>
<tr><th colspan="3">String Value</th></tr>
<tr><td>STRING</td><td>::=</td><td>\“[ˆ\t\n\”]*\”</td></tr>
<tr><td></td><td>|</td><td>[ˆ \t\n\[\]\{\}\/\”,#]*</td></tr>
</table>

Well-Formedness Constraint A string must be contained on a single line. Strings which extend over multiple lines are not permitted.

4.6 Comments

ANML supports single line comments and multi-line nested comments. Several common styles of commenting are supported for the user's convenience. For single line comments the Unix '#' and the C++ '//' comment styles are supported. Everything after a '#' or '//' on a line up until the end of the line is ignored. The following two lines demonstrate the use of the two kinds of single line comments:

```
Network[ // This is a single line comment.
```

or

```
Network[ # This is a single line comment.
```

The C-style '/* */' multi-line comments are supported with the difference that nested commenting is supported as well. The character pair '/*' marks the beginning of a comment and the matching character pair '*/' marks the end of a comment. All text between the matching beginning and ending comment symbols is ignored. For example:

```
/* This is a multi-line comment.

MyNetwork [ /* This is a nested multi-line comment. */
  _id N1
  _in_database NetBase
]

*/
```

Whereas the above comment is an error C, as nested comments are not supported, it is not an error in ANML and everything between the first occurrence of '/*' and its matching last occurrence of '*/' is ignored.

Well-formedness Constraint Comments may occur anywhere inside ANML documents except within a key word or a string value. For example, the following use of a comment causes an error in ANML:

```
Net/*Invalid comment*/work [
```

4.7 Identification System

ANML has a hierarchical identification system. The following is the syntax for a hierarchical identifier:

Hierarchical Identifier Syntax		
HIERARCHICAL_ID	::=	“.”?<LEVEL_ID>(“.”<LEVEL_ID>)*
LEVEL_ID	::=	[a-zA-Z_][a-zA-Z0-9_-]*

Every component instance must be assigned a unique ‘LEVEL_ID’ for the hierarchical level it is instantiated in. (i.e. All sub-components of a given component must all have different identifiers.) Sub-components at lower levels can then be referred to by inserting a ‘.’ in between the levels of the identifier.

Hierarchical identifiers either begin with a ‘.’ or they don’t. Identifiers which begin with a ‘.’ always refer to components starting from the same level as the component that the identifier is an attribute value of. Identifiers which don’t begin with a ‘.’ always refer to components starting from the sub-component level of the component that the identifier is an attribute value of.

For example, assume component instance with identifier ‘N’ exists. It has two subcomponents with identifiers ‘N1’ and ‘N2’. ‘N1’ and ‘N2’ component instances both have two subcomponents which have identifiers ‘H1’ and ‘H2’. Assume that ‘.N2.H1’ is an attribute value for the component instance with identifier ‘N1’. As the identifier begins with a ‘.’ it refers to a component starting from the same level of ‘N1’ component instance. Both ‘N1’ and ‘N2’ component instances are sub-components of the component instance ‘N’ and are thus at the same level. Thus ‘.N2.H1’ refers to the component instance ‘H1’ which is a sub-component of the component instance ‘N2’ which is a subcomponent of the component instance ‘N’.

Now assume that ‘N1.H2’ is an attribute value for the component instance with identifier ‘N’. As the identifier does not begin with a ‘.’ it refers to a component starting from the sub-component level of ‘N’. Thus it refers to the component instance ‘H2’ which is a sub-component of the component instance ‘N1’ which is a sub-component of the component instance ‘N’.

Attribute values of the ‘id_type’ (see Section 5.9.1.5) have the following syntax:

<table>
<tr><th colspan="3">Identifier Type Value Syntax</th></tr>
<tr><td>id_type_value</td><td>::=</td><td>HIERARCHICAL_ID</td></tr>
<tr><td></td><td>|</td><td>“{” HIERARCHICAL_ID, ... “}”</td></tr>
<tr><td></td><td>|</td><td>“[” _FROM_KEY HIERARCHICAL_ID _TO_KEY HIERARCHICAL_ID “]”</td></tr>
</table>

where ‘...’ refers to a continuation of the ‘HIERARCHICAL_ID’ value list. All values in the list must be ‘HIERARCHICAL_ID’s. The list may be empty.

When referring to a single component instance the first production above is used. The ‘HIERARCHICAL_ID’ value is the identifier of the component that is being referred to.

When referring to multiple component instances where the identifiers are not contiguous, the second production above is used. Each ‘HIERARCHICAL_ID’ in the list is an identifier of a component that is being referred to.

When referring to multiple component instances where the identifiers are contiguous, the third production above is used. The ‘HIERARCHICAL_ID’ value fro the ‘_FROM_KEY’ specifies the first identifier of the range. The ‘HIERARCHICAL_ID’ value of the ‘_TO_KEY’ specifies the last identifier of the range.

If one is instantiating components the identifiers specified using the ‘_ID_KEY’ follow the above syntax except that all identifiers must match the syntax of a ‘LEVEL_ID’ (see Section 4.4.3.4).

Validity Constraint Only the last level of the ‘HIERARCHICAL_ID’ values associate with the ‘_FROM_KEY’ and ‘_TO_KEY’ may differ.

Validity Constraint The string prefixes of the last level of the ‘HIERARCHICAL_ID’ values associated with the ‘_FROM_KEY’ and ‘_TO_KEY’ must be the same.

Validity Constraint The number suffix of the last level of the ‘HIERARCHICAL_ID’ value for the ‘_TO_KEY’ must be greater than or equal to the suffix number of the last level of the ‘HIERARCHICAL_ID’ value for the ‘_FROM_KEY’.

5 A Formal Definition of the ANML Schema

A schema is a structure that defines the allowable components and attributes of a model, and how these components may be composed together. It also specifies different constraints on attribute values and the occurrence of components. For a model to be valid the schema structure and constraints must be followed.

Some of the ideas used in the ANML Schema come from the XML Schema [10, 11]. Particularly ANML uses many of the same value and occurrence constraints.

5.1 ANML Schema Syntax

The ANML Schema syntax is actually a superset of the ANML syntax. The syntax for the ANML Schema using EBNF notation can be see in Figure 10. The terminal symbols and a few of the non-terminal symbols are not defined currently to keep the overall syntax more brief. They will be defined later in the document when each production of the syntax is explained in more detail.

schema	::=	_SCHEMA_KEY "[" schema_specs "]"
schema_specs	::=	_NAME_KEY STRING schema_spec_list
schema_spec_list	::=	schema_spec schema_spec_list | ϵ
schema_spec	::=	components | classes
components	::=	_COMPONENTS_KEY "[" component_list "]"
component_list	::=	component component_list | ϵ
component	::=	COMPONENT_NAME "[" component_specs "]"
component_specs	::=	occurs | ϵ
classes	::=	_CLASSES_KEY "[" class_list "]"
class_list	::=	class class_list | ϵ
class	::=	COMPONENT_NAME "[" class_specs "]"
class_specs	::=	_ISA_KEY STRING class_spec_list
class_spec_list	::=	class_spec class_spec_list | ϵ
class_spec	::=	_APP_CLASS_KEY STRING | _MAY_INSTANTIATE_KEY STRING | attributes | components | default
attributes	::=	_ATTRIBUTES_KEY "[" attribute_list "]"
attribute_list	::=	attribute attribute_list | ϵ
attribute	::=	ATTRIBUTE_NAME "[" attribute_specs "]"
attribute_specs	::=	_ATR_TYPE_KEY type attribute_spec_list
attribute_spec_list	::=	attribute_spec attribute_spec_list | ϵ
attribute_spec	::=	_IS_OPTIONAL_KEY STRING | constraints | attributes
default	::=	_DEFAULT_KEY "[" key_value_list "]"
occurs	::=	_OCCURS_KEY "[" occur_list "]"
occur_list	::=	occur occur_list | ϵ
constraints	::=	_CONSTRAINTS_KEY "[" constraint_list "]"
constraint_list	::=	constraint constraint_list | ϵ

Figure 10: ANML Schema Syntax

5.2 Schema and Schema Specifications

The top level non-terminal of the ANML Schema syntax is 'schema'. A schema is used to define the type of model one wishes to create. It specifies the types of components that can be contained in a model, the attributes associated with the components, and how components can be composed together. Various constraints can be specified for the occurrence of components and the value of attributes.

Schema		
schema	::=	_SCHEMA_KEY "[" schema_specs "]"
_SCHEMA_KEY	::=	"_schema"

Validity Constraint: A schema, and only one schema, must be defined within a set of ANML documents for the set of documents to be valid.

The schema specifications are given as a key-value list. The first key-value pair of this list must be the name of the schema. The name is used to help the user identify the schema. The 'STRING' value of the ' _NAME_KEY' follows the syntax of the 'STRING' terminal as given in the ANML syntax.

Schema Specifications		
schema_specs	::=	_NAME_KEY STRING schema_spec_list
_NAME_KEY	::=	"_name"

After specifying the name of the schema the remaining specifications for the schema can be given in any order.

Schema Specification List		
schema_spec_list	::=	schema_spec schema_spec_list | ϵ

There are two remaining types of schema specifications. One must specify the components that may be instantiated at the top level of the model. These are the components that may appear in the top level of the 'key_value_list' value of the 'model' production in the ANML syntax. Also, one must specify the different component classes. The component classes represent the types of components that can be contained in the model and define the attributes associated with the component and the valid sub-components.

Schema Specification		
schema_spec	::=	components | classes

Validity Constraint: 'components' specification must occur exactly once.
Validity Constraint: 'classes' specification must occur exactly once.

5.3 Components and Component Specifications

Components may be declared in two places within a schema definition. The first place is as a schema specification. Components declared as a schema specification indicate what components are allowed at the top level of a model. The second place is in a class definition. Components declared in a class definition indicate the allowable sub-components of a component of that class. The allowable components are not restricted to the declared components alone, but to all of the declared components descendants, so long as they are instantiable.

Components		
components	::=	_COMPONENTS_KEY "[" component_list "]"
_COMPONENTS_KEY	::=	"_components"

The components are declared in a key-value list.

Component List		
component_list	::=	component component_list
	|	ϵ

Validity Constraint: A component list must contain at least one component, if ‘components’ is being used as a schema specification. In order for a model to exist there must be at least one component that is allowed in the top level of a model.

A component declaration involves specifying the component name which follows the syntax of the ‘COMPONENT_NAME’ terminal in the ANML syntax and then providing any component specifications.

Component		
component	::=	COMPONENT_NAME “[” component_specs “]”

Validity Constraint: The ‘COMPONENT_NAME’ must be the name of a class defined in the same schema definition.

Along with a component declaration, occurrence constraints may be specified. The occurrence constraints restrict the allowable number of components of this type that may be instantiated at the level the ‘components’ are declared. The occurrence constraints are optional.

Component Specifications		
component_specs	::=	occurs
	|	ϵ

Validity Constraint: The ‘occurs’ specification may occur at most once.

5.4 Classes and Class Specifications

Classes are defined within a schema definition for each type of component that can be contained within a model.

Classes		
classes	::=	_CLASSES_KEY “[” class_list “]”
_CLASSES_KEY	::=	“_classes”

Classes are defined in a key-value list.

Class List		
class_list	::=	class class_list
	|	ϵ

Validity Constraint: A class list must contain at least one class definition. Without any component classes, there is nothing to compose a model with.

A class definition involves specifying the name of the class which follows the syntax of the ‘COMPONENT_NAME’ given in the ANML syntax, and then in specifying any class specifications.

Class		
class	::=	COMPONENT_NAME “[” class_specs “]”

Validity Constraint: The name (‘COMPONENT_NAME’) of the class must be unique among all other class names.

Validity Constraint: No class may have the name ‘Component’ as this is the name of the base class for all classes.

The class specifications are given as a key-value list. The first specification that must be given for a class is the parent class specification. This is done using the ‘_ISA_KEY’. The ‘STRING’ value of the ‘_ISA_KEY’ specifies the name of the parent class. All classes are derived from the base class ‘Component’. Thus if the class is not a child class of any of the defined classes it is a child class of the ‘Component’ class and ‘Component’ must be given as the value of the ‘_ISA_KEY’.

The class system for ANML is hierarchical. All classes inherit the attributes and allowable sub-components of the parent class.

Class Specifications		
class_specs	::=	_ISA_KEY STRING class_spec_list
_ISA_KEY	::=	“_isa”

Validity Constraint: The ‘STRING’ value associated with the ‘_ISA_KEY’ must be the name of a defined class or the base class ‘Component’.

Validity Constraint: No cycles may exist in the class hierarchy. i.e. A class may not be an ancestor of itself.

Validity Constraint: Each non-instantiable class must have at least one descendant class that is instantiable. **Note:** This constraint is currently not checked for.

The remaining class specifications may be given in any order.

Class Specification List		
class_spec_list	::=	class_spec class_spec_list
	|	ϵ

There are five remaining optional class specifications. One can specify the name of the application class that corresponds to the ANML component class using the ‘_APP_CLASS_KEY’. ANML has been designed so the the class hierarchy of the ANML components corresponds to the class hierarchy of the application that ANML is being used for. As the class name within the application may be different, the option to specify it is given.

One can also specify whether or not the a component of this class can be instantiated by using the ‘_MAY_INSTANTIATE_KEY’. By default, components can be instantiated for all classes, so this only needs to be used if one wishes to indicate the contrary. The ‘STRING’ value associated with the ‘_MAY_INSTANTIATE_KEY’ is a boolean value. If one wishes to indicate that components of this class may not be instantiated, then ‘false’ is entered as the ‘STRING’ value. This specification does not affect whether or not the child classes are instantiable, just the current class.

This is a useful feature if wanting to define a class which is used to categorize a set of classes, but means nothing if instantiated. In IP-TN there are different types of nodes in the topology. These include hosts, routers and hubs. Thus the no instantiation feature is used to create a common parent class for hosts, routers and hubs.

Attributes associated with this type of component, the allowable sub-components, and the default attribute values may also be specified.

Class Specification		
class_spec	::=	_APP_CLASS_KEY STRING
	|	_MAY_INSTANTIATE_KEY STRING
	|	attributes
	|	components
	|	default
_APP_CLASS_KEY	::=	“_app_class_key”
_MAY_INSTANTIATE_KEY	::=	“_may_instantiate_key”

Validity Constraint: The ‘STRING’ value associated with the ‘_MAY_INSTANTIATE_KEY’ must be a boolean value. That is it must be ‘true’ or ‘false’.

5.5 Attributes and Attribute Specifications

Attributes are defined within a class definition to specify information and characteristics which are associated with the class.

Attributes		
attributes	::=	_ATTRIBUTES_KEY “[” attribute_list “]”
_ATTRIBUTES_KEY	::=	“_attributes”

Attributes are defined in a key-value list.

Attribute List		
attribute_list	::=	attribute attribute_list
	|	ϵ

Defining an attribute involves specifying the attribute name which follows the syntax for the ‘ATTRIBUTE_NAME’ syntax, and then giving the attribute specifications.

Attribute List		
attribute	::=	ATTRIBUTE_NAME “[” attribute_specs “]”

Validity Constraint: The name (‘ATTRIBUTE_NAME’) of the attribute must be unique among all the other attribute names for a given class and its ancestor classes.

The attribute specifications are given as a key-value list. The first key-value pair of the list specifies the type of the attribute. The ’_ATR_TYPE_KEY’ is used to specify this. An attribute can be any one of the types outlined in Section 5.9.

Attribute Specifications		
attribute_specs	::=	_ATR_TYPE_KEY type attribute_spec_list
_ATR_TYPE_KEY	::=	“_atr_type”

The remaining optional attribute specifications can be given in any order.

Attribute Specification List		
attribute_spec_list	::=	attribute_spec attribute_spec_list
	|	ϵ

There are three remaining optional attribute specifications. One can specify whether the attribute is optional or not, using the ‘_IS_OPTIONAL_KEY’. By default, an attribute is mandatory, so one only need to specify this if the attribute is optional. To specify that the attribute is optional ‘true’ is given as the value for the ‘_IS_OPTIONAL_KEY’. When an attribute is marked as being optional, it means that when creating an instance of a component for which this is an attribute, the attribute does not have to be assigned. If the attribute is mandatory, the attribute must be assigned in the component instance.

If the attribute is not a composite attribute value constraints on the attribute can also be specified. If the attribute is a composite attribute, then the internal attributes can be specified.

Attribute Specification		
attribute_spec	::=	_IS_OPTIONAL_KEY STRING
	|	constraints
	|	attributes

Validity Constraint: ‘attributes’ must and may only be specified if the attribute type is ‘comp_atr’.

Validity Constraint: ‘constraints’ may only be specified if the attribute type is not ‘comp_atr’.

Validity Constraint: Only one of ‘attributes’ or ‘constraints’ may be specified.

Validity Constraint: The STRING value associated with the ‘_IS_OPTIONAL_KEY’ must be a boolean value. That is, it must be ‘true’ or ‘false’.

5.6 Default Values

In a class definition default values may be specified for the attributes and inherited attributes of a component class. Default values given in ancestor classes also apply to this class. But if a default value for an inherited attribute, which has been given a default value in an ancestor class, is given in this class, the value given in this class will be the default for this class, and the other will be ignored.

The default attribute values are given as a key-value pair list, following the syntax for ‘key_value_list’ as given in the ANML syntax. Only attributes may be assigned default values. Optional attributes should not be given default values as they would no longer be optional. The ANML Processor issues a warning in this case.

Default Values		
default	::=	_DEFAULT_KEY “[” key_value_list “]”
_DEFAULT_KEY	::=	“_default”

Validity Constraint: All keys in the default ‘key_value_list’ must be the names of attributes or inherited attributes of the class the default values are being assigned for.

Validity Constraint: Attributes that are of type ‘id_type’ may not be given default values.

5.7 Component Occurrence Constraints

Occurrence constraints can be defined within a component declaration to restrict the number of occurrences of the component and its descendant components.

Occurrence Constraints		
occurs	::=	_OCCURS_KEY "[" occur_list "]"
_OCCURS	::=	"_occurs"

Occurrence constraints are defined in a key-value pair list.

Occurrence List		
occur_list	::=	occur occur_list
	|	ϵ

There are three different occurrence constraints. One is to specify an exact number of occurrences, one to specify the minimum number of occurrences and one to specify the maximum number of occurrences. As occurrence constraints are optional, the default minimum and maximum occurrence constraints are 0 and ∞ respectively.

Occurrence Constraint		
occur	::=	_NUM_OCCUR_KEY STRING
	|	_MIN_OCCUR_KEY STRING
	|	_MAX_OCCUR_KEY STRING

5.7.1 _NUM_OCCUR_KEY

To specify that an exact number of sub-components of a certain type must be instantiated within a component or within the top level of the model, the '_NUM_OCCUR_KEY' can be used. The associated 'STRING' value is an integer which is greater than zero indicating the number of occurrences allowed.

Number Occurrence Constraint		
occur	::=	_NUM_OCCUR_KEY STRING
_NUM_OCCUR_KEY	::=	"_num_occur"

Validity Constraint: The 'STRING' value associated with the '_NUM_OCCUR_KEY' must be an integer that is greater than zero.

5.7.2 _MIN_OCCUR_KEY

To specify that a minimum number of sub-components of a certain type must be instantiated within a component or within the top level of the model, the '_MIN_OCCUR_KEY' can be used. The associated 'STRING' value is an integer which is greater than or equal to zero indicating the minimum number of occurrences allowed. By default, the minimum number of occurrences allowed is zero.

Minimum Occurrence Constraint		
occur	::=	_MIN_OCCUR_KEY STRING
_MIN_OCCUR_KEY	::=	"_min_occur"

Validity Constraint: The 'STRING' value associated with the '_MIN_OCCUR_KEY' must be an integer that is greater than or equal to zero.

Validity Constraint: The '_MIN_OCCUR_KEY' may not be defined if the 'NUM_OCCUR_KEY' has been defined.

Validity Constraint: The value of the '_MIN_OCCUR_KEY' must be less than or equal to the value of the '_MAX_OCCUR_KEY', if defined.

5.7.3 _MAX_OCCUR_KEY

To specify that a maximum number of sub-components of a certain type may be instantiated within a component or within the top level of the model, the '_MAX_OCCUR_KEY' can be used. The associated 'STRING' value is an integer which is greater than zero indicating the maximum number of occurrences allowed. By default, the maximum number of occurrences allowed is infinity.

Maximum Occurrence Constraint		
occur	::=	_MAX_OCCUR_KEY STRING
_MAX_OCCUR_KEY	::=	"_max_occur"

Validity Constraint: The 'STRING' value associated with the '_MAX_OCCUR_KEY' must be an integer that is greater than zero.

Validity Constraint: The '_MAX_OCCUR_KEY' may not be defined if the 'NUM_OCCUR_KEY' has been defined.

Validity Constraint: The value of the '_MAX_OCCUR_KEY' must be greater than or equal to the value of the '_MIN_OCCUR_KEY', if defined.

5.8 Attribute Value Constraints

Attribute value constraints can be defined within an attribute definition to restrict the allowed value of the attribute.

Attribute Value Constraints		
constraints	::=	_CONSTRAINTS_KEY "[" constraint_list "]"
_CONSTRAINT_KEY	::=	"_constraints"

Attribute value constraints are defined in a key-value pair list.

Attribute Value Constraint List		
constraint_list	::=	constraint constraint_list
	|	ϵ

There are many different types of attribute value constraints that can be defined. This allows for many different aspects of a model to be validated. The constraints that can be defined for an attribute depends on the attribute type as will be explained when each constraint is examined in more detail.

Attribute Value Constraint		
constraint	::=	_LENGTH_KEY STRING
	|	_MIN_LENGTH_KEY STRING
	|	_MAX_LENGTH_KEY STRING
	|	_NUM_ENTRIES_KEY STRING
	|	_MIN_ENTRIES_KEY STRING
	|	_MAX_ENTRIES_KEY STRING
	|	_NUM_IDS_KEY STRING
	|	_MIN_IDS_KEY STRING
	|	_MAX_IDS_KEY STRING
	|	_MIN_INCLUSIVE_KEY STRING
	|	_MIN_EXCLUSIVE_KEY STRING
	|	_MAX_INCLUSIVE_KEY STRING
	|	_MAX_EXCLUSIVE_KEY STRING
	|	_VALID_CLASSES_KEY value_list
	|	_ONE_OF_KEY value_list
	|	_PATTERN_KEY STRING

5.8.1 _LENGTH_KEY

The '_LENGTH_KEY' can be defined for the following types of attributes:

- 'string'
- 'string_list'

The 'STRING' value associated with the '_LENGTH_KEY' is an integer greater than zero that specifies the exact length the string must be. Length refers to the number of characters in the string. For the case when the attribute is of the 'string_list' type, the length restriction applies to each string in the list. That is each string in the list must be the specified length to be valid.

Length Constraint		
constraint	::=	_LENGTH_KEY STRING
_LENGTH_KEY	::=	"_length"

Validity Constraint: The 'STRING' value associated with the '_LENGTH_KEY' must be an integer that is greater than zero.

Validity Constraint: The '_LENGTH_KEY' can only be defined for attributes of type 'string' and 'string_list'.

5.8.2 _MIN_LENGTH_KEY

The '_MIN_LENGTH_KEY' can be defined for the following types of attributes:

- 'string'
- 'string_list'

The 'STRING' value associated with the '_MIN_LENGTH_KEY' is an integer greater than or equal to zero that specifies the minimum length the string must be. Length refers to the number of characters in the string. By default, the minimum allowed length of a string is zero characters. For the case when the attribute is of the 'string_list' type, the length restriction applies to each string in the list. That is each string in the list must be at least the specified length to be valid.

Minimum Length Constraint		
constraint	::=	_MIN_LENGTH_KEY STRING
_MIN_LENGTH_KEY	::=	"_min_length"

Validity Constraint: The 'STRING' value associated with the '_MIN_LENGTH_KEY' must be an integer that is greater than or equal to zero.

Validity Constraint: The '_MIN_LENGTH_KEY' may not be defined when the '_LENGTH_KEY' is defined.

Validity Constraint: The value of the '_MIN_LENGTH_KEY' must be less than or equal to the value of the '_MAX_LENGTH_KEY', if defined.

Validity Constraint: The '_MIN_LENGTH_KEY' can only be defined for attributes of type 'string' and 'string_list'.

5.8.3 _MAX_LENGTH_KEY

The '_MAX_LENGTH_KEY' can be defined for the following types of attributes:

- 'string'
- 'string_list'

The 'STRING' value associated with the '_MAX_LENGTH_KEY' is an integer greater than zero that specifies the maximum length the string may be. Length refers to the number of characters in the string. By default, the maximum allowed length of a string is infinity. For the case when the attribute is of the 'string_list' type, the length restriction applies to each string in the list. That is each string in the list may be at most the specified length to be valid.

Maximum Length Constraint		
constraint	::=	_MAX_LENGTH_KEY STRING
_MAX_LENGTH_KEY	::=	"_max_length"

Validity Constraint: The 'STRING' value associated with the '_MAX_LENGTH_KEY' must be an integer that is greater than or equal to zero.

Validity Constraint: The '_MAX_LENGTH_KEY' may not be defined when the '_LENGTH_KEY' is defined.

Validity Constraint: The value of the '_MAX_LENGTH_KEY' must be greater than or equal to the value of the '_MIN_LENGTH_KEY', if defined.

Validity Constraint: The '_MAX_LENGTH_KEY' can only be defined for attributes of type 'string' and 'string_list'.

5.8.4 _NUM_ENTRIES_KEY

The '_NUM_ENTRIES_KEY' can be defined for the following types of attributes:

- 'integer_list'
- 'real_list'
- 'boolean_list'
- 'string_list'

The 'STRING' value associated with the '_NUM_ENTRIES_KEY' is an integer greater than zero that specifies the exact number of entries that must be contained in the list.

Number of Entries Constraint		
constraint	::=	_NUM_ENTRIES STRING
_NUM_ENTRIES_KEY	::=	"_num_entries"

Validity Constraint: The 'STRING' value associated with the '_NUM_ENTRIES_KEY' must be an integer that is greater than zero.

Validity Constraint: The '_NUM_ENTRIES_KEY' can only be defined for attributes of type 'integer_list', 'real_list', 'boolean_list' and 'string_list'.

5.8.5 _MIN_ENTRIES_KEY

The '_MIN_ENTRIES_KEY' can be defined for the following types of attributes:

- 'integer_list'
- 'real_list'
- 'boolean_list'
- 'string_list'

The 'STRING' value associated with the '_MIN_ENTRIES_KEY' is an integer greater than or equal to zero that specifies the minimum number of entries that must be contained in the list. By default, the minimum number of entries allowed in a list is zero.

Minimum Entries Constraint		
constraint	::=	_MIN_ENTRIES STRING
_MIN_ENTRIES_KEY	::=	"_min_entries"

Validity Constraint: The 'STRING' value associated with the '_MIN_ENTRIES_KEY' must be an integer that is greater than or equal to zero.

Validity Constraint: The '_MIN_ENTRIES_KEY' may not be defined when the '_NUM_ENTRIES_KEY' is defined.

Validity Constraint: The value of the '_MIN_ENTRIES_KEY' must be less than or equal to the value of the '_MAX_ENTRIES_KEY', if defined.

Validity Constraint: The '_MIN_ENTRIES_KEY' can only be defined for attributes of type 'integer_list', 'real_list', 'boolean_list' and 'string_list'.

5.8.6 _MAX_ENTRIES_KEY

The '_MAX_ENTRIES_KEY' can be defined for the following types of attributes:

- 'integer_list'
- 'real_list'
- 'boolean_list'
- 'string_list'

The 'STRING' value associated with the '_MAX_ENTRIES_KEY' is an integer greater than zero that specifies the maximum number of entries that may be contained in the list. By default, the maximum number of entries allowed in a list is infinity.

Maximum Entries Constraint		
constraint	::=	_MAX_ENTRIES STRING
_MAX_ENTRIES_KEY	::=	"_max_entries"

Validity Constraint: The 'STRING' value associated with the '_MAX_ENTRIES_KEY' must be an integer that is greater than zero.

Validity Constraint: The '_MAX_ENTRIES_KEY' may not be defined when the '_NUM_ENTRIES_KEY' is defined.

Validity Constraint: The value of the '_MAX_ENTRIES_KEY' must be greater than or equal to the value of the '_MIN_ENTRIES_KEY', if defined.

Validity Constraint: The '_MAX_ENTRIES_KEY' can only be defined for attributes of type 'integer_list', 'real_list', 'boolean_list' and 'string_list'.

5.8.7 _NUM_IDS_KEY

The '_NUM_IDS_KEY' can be defined for the following types of attributes:

- 'id_type'

The 'STRING' value associated with the '_NUM_IDS_KEY' is an integer greater than zero that specifies the exact number of identifiers that must be specified in the attribute's value.

Number of Identifiers Constraint		
constraint	::=	_NUM_IDS STRING
_NUM_IDS_KEY	::=	"_num_ids"

Validity Constraint: The 'STRING' value associated with the '_NUM_IDS_KEY' must be an integer that is greater than zero.

Validity Constraint: The '_NUM_IDS_KEY' can only be defined for attributes of type 'id_type'.

5.8.8 _MIN_IDS_KEY

The '_MIN_IDS_KEY' can be defined for the following types of attributes:

- 'id_type'

The 'STRING' value associated with the '_MIN_IDS_KEY' is an integer greater than or equal to zero that specifies the minimum number of identifiers that must be specified in the attribute's value.

Minimum Identifiers Constraint		
constraint	::=	_MIN_IDS STRING
_MIN_IDS_KEY	::=	"_min_ids"

Validity Constraint: The 'STRING' value associated with the '_MIN_IDS_KEY' must be an integer that is greater than or equal to zero.

Validity Constraint: The '_MIN_IDS_KEY' may not be defined when the '_NUM_IDS_KEY' is defined.

Validity Constraint: The value of the '_MIN_IDS_KEY' must be less than or equal to the value of the '_MAX_IDS_KEY', if defined.

Validity Constraint: The '_MIN_IDS_KEY' can only be defined for attributes of type 'id_type'.

5.8.9 _MAX_IDS_KEY

The '_MAX_IDS_KEY' can be defined for the following types of attributes:

- 'id_type'

The 'STRING' value associated with the '_MAX_IDS_KEY' is an integer greater than zero that specifies the maximum number of identifiers that may be specified in the attribute's value.

Maximum Identifiers Constraint		
constraint	::=	_MAX_IDS STRING
_MAX_IDS_KEY	::=	"_max_ids"

Validity Constraint: The 'STRING' value associated with the '_MAX_IDS_KEY' must be an integer that is greater than zero.

Validity Constraint: The '_MAX_IDS_KEY' may not be defined when the '_NUM_IDS_KEY' is defined.

Validity Constraint: The value of the '_MAX_IDS_KEY' must be greater than or equal to the value of the '_MIN_IDS_KEY', if defined.

Validity Constraint: The '_MAX_IDS_KEY' may only be defined for attributes of type 'id_type'.

5.8.10 _MIN_INCLUSIVE_KEY

The '_MIN_INCLUSIVE_KEY' can be defined for the following types of attributes:

- 'integer'
- 'real'
- 'integer_list'
- 'real_list'

The 'STRING' value associated with the '_MIN_INCLUSIVE_KEY' is an integer or a real number, depending on the attribute type, that indicates the inclusive minimum value allowed for the attribute. That is, the value of the attribute must be greater than or equal to the inclusive minimum specified. For attributes that are of the 'integer_list' or 'real_list' type, the inclusive minimum applies to all numbers in the list. That is, each number in the list must be greater than or equal to the inclusive minimum value.

Minimum Inclusive Constraint		
constraint	::=	_MIN_INCLUSIVE STRING
_MIN_INCLUSIVE_KEY	::=	"_min_inclusive"

Validity Constraint: The 'STRING' value associated with the '_MIN_INCLUSIVE_KEY' must be an integer value if the attribute is of the 'integer' or 'integer_list' types.

Validity Constraint: The 'STRING' value associated with the '_MIN_INCLUSIVE_KEY' must be a real value if the attribute is of the 'real' or 'real_list' types.

Validity Constraint: The '_MIN_INCLUSIVE_KEY' may not be defined if the '_MIN_EXCLUSIVE_KEY' is defined.

Validity Constraint: The value of the '_MIN_INCLUSIVE_KEY' must be less than or equal to the value of the '_MAX_INCLUSIVE_KEY', if defined.

Validity Constraint: The value of the '_MIN_INCLUSIVE_KEY' must be less than the value of the '_MAX_EXCLUSIVE_KEY', if defined.

Validity Constraint: The '_MIN_INCLUSIVE_KEY' may only be defined for attributes of type 'integer', 'real', 'integer_list' and 'real_list'.

5.8.11 _MIN_EXCLUSIVE_KEY

The '_MIN_EXCLUSIVE_KEY' can be defined for the following types of attributes:

- 'integer'
- 'real'

- ‘integer_list’
- ‘real_list’

The ‘STRING’ value associated with the ‘_MIN_EXCLUSIVE_KEY’ is an integer or a real number, depending on the attribute type, that indicates the exclusive minimum value allowed for the attribute. That is, the value of the attribute must be greater than the exclusive minimum specified. For attributes that are of the ‘integer_list’ or ‘real_list’ type, the exclusive minimum applies to all numbers in the list. That is, each number in the list must be greater than the exclusive minimum value.

Minimum Exclusive Constraint		
constraint	::=	_MIN_EXCLUSIVE STRING
_MIN_EXCLUSIVE_KEY	::=	“_min_exclusive”

Validity Constraint: The ‘STRING’ value associated with the ‘_MIN_EXCLUSIVE_KEY’ must be an integer value if the attribute is of the ‘integer’ or ‘integer_list’ types.

Validity Constraint: The ‘STRING’ value associated with the ‘_MIN_EXCLUSIVE_KEY’ must be a real value if the attribute is of the ‘real’ or ‘real_list’ types.

Validity Constraint: The ‘_MIN_EXCLUSIVE_KEY’ may not be defined if the ‘_MIN_INCLUSIVE_KEY’ is defined.

Validity Constraint: The value of the ‘_MIN_EXCLUSIVE_KEY’ must be less than the value of the ‘_MAX_INCLUSIVE_KEY’, if defined.

Validity Constraint: The value of the ‘_MIN_EXCLUSIVE_KEY’ must be less than the value of the ’_MAX_EXCLUSIVE_KEY’, if defined.

Validity Constraint: The ‘_MIN_EXCLUSIVE_KEY’ may only be defined for attributes of type ‘integer’, ‘real’, ‘integer_list’ and ‘real_list’.

5.8.12 _MAX_INCLUSIVE_KEY

The ‘_MAX_INCLUSIVE_KEY’ can be defined for the following types of attributes:

- ‘integer’
- ‘real’
- ‘integer_list’
- ‘real_list’

The ‘STRING’ value associated with the ‘_MAX_INCLUSIVE_KEY’ is an integer or a real number, depending on the attribute type, that indicates the inclusive maximum value allowed for the attribute. That is, the value of the attribute must be less than or equal to the inclusive maximum specified. For attributes that are of the ‘integer_list’ or ‘real_list’ type, the inclusive maximum applies to all numbers in the list. That is, each number in the list must be less than or equal to the inclusive maximum value.

Maximum Inclusive Constraint		
constraint	::=	_MAX_INCLUSIVE STRING
_MAX_INCLUSIVE_KEY	::=	“_max_inclusive”

Validity Constraint: The ‘STRING’ value associated with the ‘_MAX_INCLUSIVE_KEY’ must be an integer value if the attribute is of the ‘integer’ or ‘integer_list’ types.

Validity Constraint: The ‘STRING’ value associated with the ‘_MAX_INCLUSIVE_KEY’ must be a real value if the attribute is of the ‘real’ or ‘real_list’ types.

Validity Constraint: The ‘_MAX_INCLUSIVE_KEY’ may not be defined if the ‘_MAX_EXCLUSIVE_KEY’ is defined.

Validity Constraint: The value of the ‘_MAX_INCLUSIVE_KEY’ must be greater than or equal to the value of the ‘_MIN_INCLUSIVE_KEY’, if defined.

Validity Constraint: The value of the ‘_MAX_INCLUSIVE_KEY’ must be greater than the value of the ’_MIN_EXCLUSIVE_KEY’, if defined.

Validity Constraint: The ‘_MAX_INCLUSIVE_KEY’ may only be defined for attributes of type ‘integer’, ‘real’, ‘integer_list’ and ‘real_list’.

5.8.13 _MAX_EXCLUSIVE_KEY

The '_MAX_EXCLUSIVE_KEY' can be defined for the following types of attributes:

- 'integer'
- 'real'
- 'integer_list'
- 'real_list'

The 'STRING' value associated with the '_MAX_EXCLUSIVE_KEY' is an integer or a real number, depending on the attribute type, that indicates the exclusive maximum value allowed for the attribute. That is, the value of the attribute must be less than the exclusive maximum specified. For attributes that are of the 'integer_list' or 'real_list' type, the exclusive maximum applies to all numbers in the list. That is, each number in the list must be less than the exclusive maximum value.

Maximum Exclusive Constraint		
constraint	::=	_MAX_EXCLUSIVE STRING
_MAX_EXCLUSIVE_KEY	::=	"_max_exclusive"

Validity Constraint: The 'STRING' value associated with the '_MAX_EXCLUSIVE_KEY' must be an integer value if the attribute is of the 'integer' or 'integer_list' types.

Validity Constraint: The 'STRING' value associated with the '_MAX_EXCLUSIVE_KEY' must be a real value if the attribute is of the 'real' or 'real_list' types.

Validity Constraint: The '_MAX_EXCLUSIVE_KEY' may not be defined if the '_MAX_INCLUSIVE_KEY' is defined.

Validity Constraint: The value of the '_MAX_EXCLUSIVE_KEY' must be greater than the value of the '_MIN_INCLUSIVE_KEY', if defined.

Validity Constraint: The value of the '_MAX_EXCLUSIVE_KEY' must be greater than the value of the '_MIN_EXCLUSIVE_KEY', if defined.

Validity Constraint: The '_MAX_EXCLUSIVE_KEY' may only be defined for attributes of type 'integer', 'real', 'integer_list' and 'real_list'.

5.8.14 _VALID_CLASSES_KEY

The '_VALID_CLASSES_KEY' can be defined for the following types of attributes:

- 'id_type'

The 'value_list' associated with the '_VALID_CLASSES_KEY' is a list of the classes that the components, referred to by the identifiers specified in the value of the attribute, may be instances of. (The 'value_list' referred to here is a non-terminal in the ANML syntax.) Putting a class in this list automatically includes all descendant classes as well.

Valid Classes Constraint		
constraint	::=	_VALID_CLASSES value_list
_VALID_CLASSES_KEY	::=	"_valid_classes"

Validity Constraint: Each entry of the 'value_list' associated with the '_VALID_CLASSES_KEY' must be the name of a class that has been defined in the schema definition.

Validity Constraint: The '_VALID_CLASSES_KEY' can only be defined for attributes of type 'id_type'.

5.8.15 _ONE_OF_KEY

The '_ONE_OF_KEY' can be defined for the following types of attributes:

- 'integer'
- 'real'
- 'string'
- 'integer_list'

- ‘real_list’
- ‘string_list’

The ‘value_list’ associated with the ‘_ONE_OF_KEY’ is a list of allowable values for the attribute. (The ‘value_list’ referred to here is a non-terminal in the ANML syntax.) This means that the value of the attribute must be one of the values specified by the ‘_ONE_OF_KEY’. For the case when the attribute is of one of the list types, each value in the list must be one of the values specified by the ‘_ONE_OF_KEY’.

One Of Constraint		
constraint	::=	_ONE_OF_KEY value_list
_ONE_OF_KEY	::=	“_one_of”

Validity Constraint: Each entry of the ‘value_list’ associated with the ‘_ONE_OF_KEY’ must be an integer if the attribute type is ‘integer’ or ‘integer_list’.

Validity Constraint: Each entry of the ‘value_list’ associated with the ‘_ONE_OF_KEY’ must be a real number if the attribute type is ‘real’ or ‘real_list’.

Validity Constraint: Each entry of the ‘value_list’ associated with the ‘_ONE_OF_KEY’ must be a string if the attribute type is ‘string’ or ‘string_list’.

Validity Constraint: If ‘_ONE_OF_KEY’ is defined, no other attribute value constraints may be defined except for ‘_NUM_ENTRIES_KEY ‘_MIN_ENTRIES_KEY’ and ‘_MAX_ENTRIES_KEY’.

Validity Constraint: The ‘_ONE_OF_KEY’ can only be defined for attributes of type ‘id_type’.

5.8.16 _PATTERN_KEY

The ‘_PATTERN_KEY’ can be defined for the following types of attributes:

- ‘string’
- ‘string_list’

The ‘STRING’ value associated with the ‘PATTERN_KEY’ is a regular expression that the value of the attribute must match in order to be valid. For the case when the attribute is of the ‘string_list’, each string in the list must match the specified regular expression.

Pattern		
constraint	::=	_PATTERN_KEY STRING
_PATTERN_KEY	::=	“_pattern”

Validity Constraint: The ‘_PATTERN_KEY’ can only be defined for attributes of type ‘string’ and ‘string_list’.

Note: The ‘_PATTERN_KEY’ is currently not supported.

5.9 Attribute Types

Every attribute must be assigned a certain type, to help identify and restrict the attribute value contents. Attribute types are divided into two main categories: primitive types and compound types.

Attribute Types		
type	::=	prim_type
	|	comp_type

5.9.1 Primitive Attribute Types

Primitive types are the basic types which are indivisible.

Primitive Attribute Types		
prim_type	::=	INTEGER_TYPE
	|	REAL_TYPE
	|	BOOLEAN_TYPE
	|	STRING_TYPE
	|	ID_TYPE

5.9.1.1 INTEGER_TYPE

One of the primitive attribute types is the 'INTEGER_TYPE'. The value of an attribute of this type must be a single integer, as defined mathematically.

Integer Type		
INTEGER_TYPE	::=	"integer"

An integer must match the following regular expression.

Regular Expression for Integer		
INTEGER	::=	[-+]?[0-9]+

The following constraints can be defined for attributes of the 'integer' type:

- '_MIN_INCLUSIVE_KEY'
- '_MIN_EXCLUSIVE_KEY'
- '_MAX_INCLUSIVE_KEY'
- '_MAX_EXCLUSIVE_KEY'
- '_ONE_OF_KEY'

5.9.1.2 REAL_TYPE

Another primitive attribute type is the 'REAL_TYPE'. The value of an attribute of this type must be a single real number as defined mathematically.

Real Type		
REAL_TYPE	::=	"real"

A real number must match the following regular expression.

Regular Expression for Real Number		
REAL	::=	[-+]?[0-9]+("."[0-9]+)?([eE][-+]?[0-9]+)?

The following constraints can be defined for attributes of the 'integer' type:

- '_MIN_INCLUSIVE_KEY'
- '_MIN_EXCLUSIVE_KEY'
- '_MAX_INCLUSIVE_KEY'
- '_MAX_EXCLUSIVE_KEY'
- '_ONE_OF_KEY'

5.9.1.3 BOOLEAN_TYPE

Another primitive attribute type is the 'BOOLEAN_TYPE'. The value of an attribute of this type indicates that something is either true or false.

Boolean Type		
BOOLEAN_TYPE	::=	"boolean"

A boolean value must match the following syntax.

Boolean Value Syntax		
BOOLEAN	::=	'true'
	|	'false'

No constraints may be defined for attributes of the 'boolean' type. A boolean value is simply 'true' or 'false'.

5.9.1.4 STRING_TYPE

Another primitive attribute type is the 'STRING_TYPE'. The value of an attribute of this type is a string of characters.

String Type		
STRING_TYPE	::=	"string"

A string value follows the syntax of the 'STRING' terminal defined in the ANML syntax. See Figure 9.
The following constraints can be defined for attributes of the 'string' type:

- '_LENGTH_KEY'
- '_MIN_LENGTH_KEY'
- '_MAX_LENGTH_KEY'
- '_ONE_OF_KEY'
- '_PATTERN_KEY'

5.9.1.5 ID_TYPE

Another primitive attribute type is the 'ID_TYPE'. The 'ID_TYPE' is the most complex of all the primitive datatypes, but is given it's own unique datatype to serve the special purpose of identifying different components within a model, and to validate that the component exists.

ID Type		
ID_TYPE	::=	"id_type"

Attributes of the 'id_type' must match the following syntax:

Id Type Value Syntax		
id_type_value	::=	HIERARCHICAL_ID
	|	"{" HIERARCHICAL_ID, ... "}"
	|	"[" _FROM_KEY HIERARCHICAL_ID _TO_KEY HIERARCHICAL_ID "]"

Thus the value of an attribute of the 'id_type' may be a single identifier, a list of identifiers, or a range of identifiers. More information on the identifiers and their syntax can be seen in Section 4.7.

Part of validating an attribute value of the 'id_type', besides any constraints which have been specified, is checking to make sure that a components with the given identifiers actually exist.

The following constraints can be defined for attributes of the 'id_type':

- '_NUM_IDS_KEY'
- '_MIN_IDS_KEY'
- '_MAX_IDS_KEY'
- '_VALID_CLASSES_KEY'

5.9.2 Compound Attribute Types

Compound types are types which are built on primitive types.

Compound Attribute Types		
comp_type	::=	INTEGER_LIST_TYPE
	|	REAL_LIST_TYPE
	|	BOOLEAN_LIST_TYPE
	|	STRING_LIST_TYPE
	|	COMP_ATR_TYPE

5.9.2.1 INTEGER_LIST_TYPE

One compound attribute type is the 'INTEGER_LIST_TYPE'. An attribute of this type has a value which is a list of values of the 'INTEGER_TYPE'.

Integer List Type		
INTEGER_LIST_TYPE	::=	"integer_list"

Attributes of the 'integer_list_type' must match the following syntax:

Integer List Syntax		
integer_list_syntax	::=	"" value_list ""

where each entry in the list must be an 'INTEGER'.

The following constraints can be defined for attributes of the 'integer_list' type:

- '_NUM_ENTRIES_KEY'
- '_MIN_ENTRIES_KEY'
- '_MAX_ENTRIES_KEY'
- '_MIN_INCLUSIVE_KEY'
- '_MIN_EXCLUSIVE_KEY'
- '_MAX_INCLUSIVE_KEY'
- '_MAX_EXCLUSIVE_KEY'
- '_ONE_OF_KEY'

5.9.2.2 REAL_LIST_TYPE

Another compound attribute type is the 'REAL_LIST_TYPE'. An attribute of this type has a value which is a list of values of the 'REAL_TYPE'.

Real List Type		
REAL_LIST_TYPE	::=	"real_list"

Attributes of the 'real_list_type' must match the following syntax:

Real List Syntax		
real_list_syntax	::=	"" value_list ""

where each entry in the list must be an 'REAL' number.

The following constraints can be defined for attributes of the 'real_list' type:

- '_NUM_ENTRIES_KEY'
- '_MIN_ENTRIES_KEY'
- '_MAX_ENTRIES_KEY'
- '_MIN_INCLUSIVE_KEY'
- '_MIN_EXCLUSIVE_KEY'
- '_MAX_INCLUSIVE_KEY'
- '_MAX_EXCLUSIVE_KEY'
- '_ONE_OF_KEY'

5.9.2.3 BOOLEAN_LIST_TYPE

Another compound attribute type is the 'BOOLEAN_LIST_TYPE'. An attribute of this type has a value which is a list of values of the 'BOOLEAN_TYPE'.

Boolean List Type		
BOOLEAN_LIST_TYPE	::=	"boolean_list"

Attributes of the 'boolean_list_type' must match the following syntax:

Boolean List Syntax		
boolean_list_syntax	::=	"" value_list ""

where each entry in the list must be a 'BOOLEAN'.

The following constraints can be defined for attributes of the 'boolean_list' type:

- '_NUM_ENTRIES_KEY'
- '_MIN_ENTRIES_KEY'
- '_MAX_ENTRIES_KEY'

5.9.2.4 STRING_LIST_TYPE

Another compound attribute type is the 'STRING_LIST_TYPE'. An attribute of this type has a value which is a list of values of the 'STRING_TYPE'.

String List Type		
STRING_LIST_TYPE	::=	"string_list"

Attributes of the 'string_list_type' must match the following syntax:

String List Syntax		
string_list_syntax	::=	"" value_list ""

where each entry in the list must be a 'STRING'.

The following constraints can be defined for attributes of the 'string_list' type:

- '_NUM_ENTRIES_KEY'
- '_MIN_ENTRIES_KEY'
- '_MAX_ENTRIES_KEY'
- '_LENGTH_KEY'
- '_MIN_LENGTH_KEY'
- '_MAX_LENGTH_KEY'
- '_ONE_OF_KEY'
- '_PATTERN_KEY'

5.9.2.5 COMP_ATR_TYPE

Another compound attribute type is the 'COMP_ATR_TYPE' or composite attribute type. An attribute of this type is an attribute which is composed of inner attributes.

Composite Attribute Type		
COMP_ATR_TYPE	::=	"comp_atr"

Attributes of the 'comp_atr_type' must match the following syntax:

String List Syntax		
string_list_syntax	::=	"[" key_value_list "]"

where each key-value pair must be one of the defined inner attributes.

No constraints may be defined for attributes of the 'comp_atr_type'. Constraints may be defined for inner attributes though, so long as they are not of the 'comp_atr_type' themselves.

6 Processing ANML Models

ANML models are intended to be used as input to a software module called an ‘application’. The application may be a simulator, a graphical tool to display the model, a tool to translate the ANML model into some other format, or some other program.

In order to process an ANML model, a software module called the ‘ANML Processor’ is first used to check that the model is well-formed and valid. The application may then take the information provided by the ANML Processor and further process or utilize the information as desired.

The ANML Processor is independent of the model type and can be used by any application. The ANML Processor generates errors for any well-formedness constraints, validity constraints or other constraints imposed by the schema which are not satisfied.

References

[1] Cowie, J., H. Liu, J. Liu, D. Nicol, and A. Ogielski. 1999. Towards realistic million-node Internet simulations. In *Proceedings of 1999 International Conference on Parallel and Distributed Processing Techniques and Applications.* (PDPTA ’99)

[2] Cowie, J. H., D. M. Nicol, and A. T. Ogielski. 1999. Modeling 100,000 nodes and beyond: self-validating design. *DARPA/NIST Workshop on Validation of Large Scale Network Simulation Models.*

[3] Cowie, J. H., D. M. Nicol, and A. T. Ogielski. 1999. Modeling the global Internet. *Computing in Science & Engineering 1(1):42-50.* (January/February 1999)

[4] Himsolt, M. 1996a. GML: a portable graph file format. *University of Passau Technical Report*

[5] Himsolt, M. 1996b. GML: Graph Modelling Language manual. University of Passau. http://infosun.fmi.uni-passau.de/Graphlet/GML/index.html

[6] Levine, J. R., T. Mason, and D. Brown. 1992. lex & yacc, 28–29. O’Reilly & Associates, Inc.

[7] Ogielski, A. T. Domain Modeling Language (DML) Reference Manual, 1999.

[8] Unger, B., F. Gomes, Z. Xiao, P. Gburzynski, T. Ono-Tesfaye, S. Ramaswamy, C. Williamson, and A. Covington. 1995. A high fidelity ATM traffic and network simulator. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, 996–1003. The Society for Computer Simulation, International, San Diego, California.

[9] W3C - World Wide Web Consortium. 1998. Extensible Markup Language (XML) 1.0 - W3C Recommendation 10-February-1998.

[10] W3C - World Wide Web Consortium. 2000. XML Schema Part 1:Structures. W3C Working Draft 7 April 2000.

[11] W3C - World Wide Web Consortium. 2000. XML Schema Part 2:Datatypes. W3C Working Draft 7 April 2000.

A ANML Files Used in Tutorials

In this appendix the ANML files that were used in the ANML and ANML Schema tutorials are given in full.

```
/*============================================================================
 * File: net_schema.nml
 *
 * Description: This file contains a basic schema definition for network models.
 *============================================================================
 */

_schema [
  _name NetSchema

  _components [
    /* Allow only a single Network instance at top-level
     */
    Network [ _occurs [ _num_occur 1] ]

  ] // end _schema _components

  _classes [

    /*-------------------------------------------------------------------------
     * Class: Network
     *
     * Description: This is a class used to represent a network. A network
     *              may be further composed of other networks and can contain
     *              different network components.
     *
     * Instantiable: yes
     *-------------------------------------------------------------------------
     */
    Network[
      _isa Component
      _components[
        /* A Network may contain subnets */
        Network []

        /* A Network may contain different nodes */
        Node []

        /* A Network may contain links to connect the nodes together */
        Link []

      ] // end Network _components

    ] // end Network

    /*-------------------------------------------------------------------------
     * Class: Node
     *
     * Description: This is a class to represent the different nodes that can
     *              exist on network such as routers and hosts.
     *
```

```
 * Instantiable: no
 *-------------------------------------------------------------------------
 */
Node[
  _isa Component
  _may_instantiate false

] // end Node

/*-------------------------------------------------------------------------
 * Class: Link
 *
 * Description: This class is a base class for the different links
 *              that connect nodes on a network together.
 *
 * Instantiable: no
 *-------------------------------------------------------------------------
 */
Link[
  _isa Component
  _may_instantiate false

  _attributes[

    /*.....................................................................
     * Attribute: delay
     *
     * Type: real
     *
     * Description: - the propagation delay on the link in seconds.
     *
     * Use: Required
     *
     * Constraints: - 0.0 < prop_delay
     *.....................................................................
     */
    delay [
      _atr_type real
      _constraints [ _min_exclusive 0.0 ]
    ]

    /*.....................................................................
     * Attribute: rate
     *
     * Type: real
     *
     * Description: - the rate of the link in Mbps
     *
     * Use: Required
     *
     * Constraints: - 0 < rate
     *.....................................................................
     */
    rate [
```

```
    _atr_type real
    _constraints [ _min_exclusive 0]
  ]

  /*.....................................................................
   * Attribute: mtu
   *
   * Type: integer
   *
   * Description: - the maximum transmission unit for the link
   *               (i.e. the largest size of a packet in bytes that
   *                can be transmitted across the link)
   *
   * Use: Required
   *
   * Constraints: - 68 <= mtu <= 65535
   *.....................................................................
   */
  mtu [
    _atr_type integer
    _constraints [ _min_inclusive 68 _max_inclusive 65535 ]
  ]

 ] // end Link _attributes

] // end Link

/*---------------------------------------------------------------------------
 * Class: Router
 *
 * Description: This is a class to represent a router. A router is a node
 *              on a network that determines which neighboring node a
 *              packet should be sent to on its way to its destination.
 *
 * Instantiable: yes
 *---------------------------------------------------------------------------
 */
Router[
  _isa Node

  _attributes [

    /*.....................................................................
     * Attribute: proc_delay
     *
     * Type: real
     *
     * Description: - the time it takes to process a packet in the router in
     *                seconds.
     *
     * Use: Default - default of 0.000001 seconds given in _default section
     *              - default is overridden if user defines attribute
     *
     * Constraints: - 0 < proc_delay
```

```
 *...................................................................
 */
proc_delay [
  _atr_type real
  _constraints [ _min_exclusive 0 ]
]

/*...................................................................
 * Attribute: buffer_size
 *
 * Type: integer
 *
 * Description: - size of buffer on each router interface in Bytes.
 *
 * Use: Default - default of 50 kB given in _default section
 *              - default is overridden if user defines attribute
 *
 * Constraints: - 0 < buffer_size
 *...................................................................
 */
buffer_size [
  _atr_type integer
  _constraints [ _min_exclusive 0 ]
]

/*...................................................................
 * Attribute: lan_links
 *
 * Type: id_type
 *
 * Description: - a router may be on zero or more LANs. This attribute
 *                is specified to indicate the LAN_Links that the
 *                router is attached to.
 *
 * Use: Optional
 *
 * Constraints: - id must refer to a component of the LAN_Link class
 *...................................................................
 */
lan_links [
  _atr_type id_type
  _is_optional true
  _constraints [ _valid_classes {LAN_Link} ]
]

] // end Router _attributes

_default[

  proc_delay 0.000001
  buffer_size 51200 // 51200 Bytes = 50 kB

] // end Router _default
```

```
] // end Router

/*-------------------------------------------------------------------------
 * Class: Host
 *
 * Description: This is a class representing a host. In this model hosts
 *              are considered to be end nodes on a network, not
 *              responsible for routing, that are generate or
 *              receive traffic of some sort.
 *
 * Instantiable: yes
 *------------------------------------------------------------------------
 */
Host[
  _isa Node

  _attributes[

    /*.................................................................
     * Attribute: buffer_size
     *
     * Type: integer
     *
     * Description: - size of buffer on host interface in Bytes.
     *
     * Use: Default - default of 50 kB given in _default section
     *              - default is overridden if user defines attribute
     *
     * Constraints: - 0 < buffer_size
     *.................................................................
     */
    buffer_size [
      _atr_type integer
      _constraints [ _min_exclusive 0 ]
    ]


    /*.................................................................
     * Attribute: lan_link
     *
     * Type: id_type
     *
     * Description: - As a host is considered to be an end node for this
     *                network model, and may not route packets, it may
     *                be connected to at most one LAN. (A Host could be
     *                on a point-to-point network.)
     *
     * Use: Optional
     *
     * Constraints: - id must refer to a component of the LAN_Link class
     *.................................................................
     */
    lan_link [
      _atr_type id_type
```

```
      _is_optional true
      _constraints [ _valid_classes {LAN_Link} _max_ids 1 ]
    ]

  ] // end BasicHost _attributes

  _default[

    buffer_size 51200 // 51200 Bytes = 50 kB

  ] // end BasicHost _default

] // end BasicHost

/*--------------------------------------------------------------------------
 * Class: P2P_Link
 *
 * Description: This is a class representing a point-to-point link between
 *              two nodes.
 *
 * Instantiable: yes
 *--------------------------------------------------------------------------
 */
P2P_Link[
  _isa Link

  _attributes[

    /*.....................................................................
     * Attribute: nodeA
     *
     * Type: id_type
     *
     * Description: - the node on one end of the link
     *
     * Use: Required
     *
     * Constraints: - id must refer to component which is a member of
     *                Node class
     *              - only one id may be given
     *.....................................................................
     */
    nodeA [
      _atr_type id_type
      _constraints [ _valid_classes {Node} _num_ids 1 ]
    ]

    /*.....................................................................
     * Attribute: nodeB
     *
     * Type: id_type
     *
     * Description: - the node on the other end of the link
     *
```

```
     * Use: Required
     *
     * Constraints: - id must refer to component which is a member of
     *                Node class
     *              - only one id may be given
     *.................................................................
     */
    nodeB [
      _atr_type id_type
      _constraints [ _valid_classes {Node} _num_ids 1 ]
    ]

  ] // end P2P_Link _attributes

  _default[
    mtu 9180  /* have a default mtu of 9180 bytes */
  ]

] // end P2P_Link

/*---------------------------------------------------------------------------
 * Class: LAN_Link
 *
 * Description: Represents a link to join nodes on a LAN together.
 *
 * Instantiable: yes
 *---------------------------------------------------------------------------
 */
LAN_Link[
  _isa Link

  _default[
    mtu 1500  /* have a default mtu of 1500 bytes - Ethernet Standard */
  ]

] // end LAN_Link

] // end _schema _classes

] // end _schema
```

```
/*==============================================================================
 * File: net_databases.nml
 *
 * Description: This file contains different databases used to define
 *              different components for use in network models.
 *==============================================================================
 */

/*------------------------------------------------------------------------------
 * Database: Node_DB
 *
 * Description: Contains definitions of different network nodes.
 *------------------------------------------------------------------------------
 */
_database [
  _name Node_DB

  StdHost[ _class Host buffer_size 102400 ]

  StdRouter[ _class Router proc_delay 0.000005 buffer_size 102400 ]

] // end _database Node_DB

/*------------------------------------------------------------------------------
 * Database: Link_DB
 *
 * Description: Contains definitions of different links.
 *------------------------------------------------------------------------------
 */
_database [
  _name Link_DB

  /* 10Mbps LAN_Link (representative of 10Mbps Ethernet) */
  LAN_10Mbps[ _class LAN_Link rate 10]

  /* an OC-3 point-to-point link - 155.52 Mbps */
  Link_OC3[ _class P2P_Link rate 155.52 ]

] // end _database Link_DB

/*------------------------------------------------------------------------------
 * Database: Network_DB
 *
 * Description: Contains definitions of various networks.
 *------------------------------------------------------------------------------
 */
_database [
  _name Network_DB

  /* A LAN with two hosts */
  Net2H[
    _class Network
    LAN_10Mbps[ _id L1 _in_database Link_DB delay 0.00001]
    StdHost[ _id {H1,H2} _in_database Node_DB lan_link .L1]
```

```
  ]

  /* A LAN with three hosts */
  Net3H[
    _class Network
    LAN_10Mbps[ _id L1 _in_database Link_DB delay 0.00001]
    StdHost[ _id [_from H1 _to H3] _in_database Node_DB lan_link .L1]
  ]

  /* A network with two subnets, both having two hosts each. A router
   * joins the two subnets.
   */
  Net2S_A[
    _class Network
    Net2H[ _id {N1,N2} _in_database Network_DB]
    StdRouter[ _id R1 _in_database Node_DB lan_links{.N1.L1, .N2.L1} ]
  ]

  /* A network with two subnets, one with two hosts, and one with three hosts.
   * A router joins the two subnets.
   */
  Net2S_B[
    _class Network
    Net2H[ _id N1 _in_database Network_DB]
    Net3H[ _id N2 _in_database Network_DB]
    StdRouter[ _id R1 _in_database Node_DB lan_links{.N1.L1, .N2.L1} ]
  ]

] // end _database Network_DB
```

```
/*==============================================================================
 * File: tut1_model.anml
 *
 * Description: This file describes a network model that consists of three
 *              main networks. The three main networks are composed of
 *              further subnets and are joined by point-to-point links.
 *==============================================================================
 */

_include net_schema.anml
_include net_databases.anml

_model[
  _name tut1_model
  _use_schema NetSchema

  Network[
    _id Netmodel

    /* The model is comprised of three main networks */
    Net2S_B[ _id {N1,N3} _in_database Network_DB ]
    Net2S_A[ _id N2 _in_database Network_DB ]

    /* The links to join the three networks together */
    Link_OC3[ _id L1 _in_database Link_DB
              delay 0.0001 nodeA .N1.R1 nodeB .N3.R1]
    Link_OC3[ _id L2 _in_database Link_DB
              delay 0.0002 nodeA .N1.R1 nodeB .N2.R1]
    Link_OC3[ _id L3 _in_database Link_DB
              delay 0.00015 nodeA .N2.R1 nodeB .N3.R1]

  ]

]
```