The MUSKOX Concepts Guide

VERSION 1.2

MASTER Information Systems Corporation

MASTERSYS

The MUSKOX Concepts Guide
Version 1.2

January, 1996

Master Information Systems Corporation
3596 Pimlico Drive
Pleasanton,
CA 94588

http://misc-sun.mastersys.com
master@mastersys.com

Copyright © 1995, 1996 **Master** Information Systems Corporation

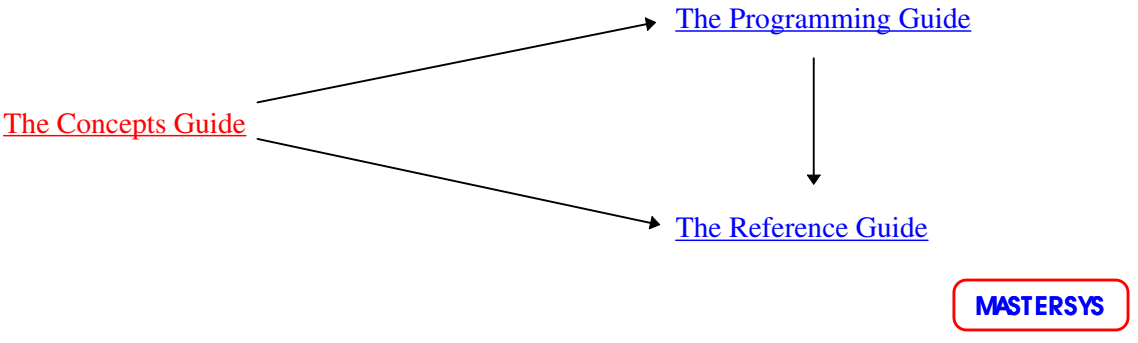
This document contains proprietary information of **Master** Information Systems Corporation (**MASTERSYS**). It is provided under a license agreement and is also protected by copyright law. Permission to copy is granted provided that the copies are not made or distributed for direct commercial advantage.

The information in this document is subject to change without notice. **MASTERSYS** does not warrant that this document is error-free.

MUSKOX, MASTERSYS are trademarks of **Master** Information Systems Corporation. All other trademarks belong to their respective owners.

ONLINE DOCUMENTATION ROADMAP

You can view this and other MUSKOX Guides online using free ACROBAT READER from http://www.adobe.com. Follow red hypertext links within a document, and blue links between different documents. Usually blue links will guide you in this order:



CONTENTS

1. INTRODUCTION 1

1.1 STATES 1
1.1.1 States in Our World 1
1.1.2 Behaviour -- Changing States According to Laws 1
1.1.3 Understanding States 1
1.1.4 States, Computers, and Software Engineering 2
1.1.5 Programming Computers with States Using MUSKOX 2
1.2 INTENDED AUDIENCE AND PURPOSE 2
1.3 TYPOGRAPHICAL CONVENTIONS 2
1.4 MUSKOX PRODUCT OVERVIEW 3
1.5 APPLICATION DEVELOPMENT WITH MUSKOX 3
1.6 MUSKOX DOCUMENTATION 4
1.7 ORGANIZATION OF THE MUSKOX CONCEPTS GUIDE 4

2. FUNDAMENTALS OF THE MUSKOX METALANGUAGE 5

2.1 PROGRAMMING WITH STATES 5
2.1.1 States in Traditional Programming 5
2.1.2 Declaring States with the MUSKOX Metalanguage 6
2.2 OBJECTS AND OBJECT LIFECYCLES 6
2.3 STATES AND STATE TRANSITION RULES 6
2.3.1 Combining State Transition Rules 7
2.4 MESSAGES 8
2.5 SYMBOLIC COMMUNICATION 8
2.6 NON-BLOCKING QUEUEING MECHANISM 9
2.7 THE STATE TRANSITION ENGINE 9
2.8 ACTIONS 11
2.9 CONCURRENCY 11
2.10 THE MUSKOX OBJECT MODEL 11
2.10.1 The MUSKOX Type 11
2.10.2 The MUSKOX Object 12
2.10.3 Object Lifetime 12
2.10.4 Links and Communication 12
2.10.5 Synchronization of Lifecycles 13
2.11 PROGRAMS AND LIFECYCLES 13
2.11.1 Differences 13
2.11.2 Similarities 13
2.12 CONCLUSION 14

3. THE MUSKOX COMPILER 15

3.1 MUSKOX SPECS 15
3.2 FILES GENERATED BY THE MUSKOX COMPILER 15
3.3 MUSKOX AND THE HOST PROGRAMMING LANGUAGE C++ 16
3.4 MAPPING TO THE HOST LANGUAGE C++ 16

4. THE MUSKOX APPLICATION RUNTIME MONITOR (ARM) 19

4.1 BUILDING THE ARM 19
4.2 THE FRAMEWORK 19
4.2.1 Standard Framework 20
4.3 THE STATE TRANSITION ENGINE 20

5. MUSKOX APPLICATIONS 21

5.1 HIGH-AVAILABILITY SYSTEMS 21
5.1.1 MUSKOX and High-Availability Systems 22
5.1.2 Advantages of Using MUSKOX in High-Availability Systems 24
5.2 MUSKOX AND OBJECT-ORIENTED FRAMEWORKS 24
5.2.1 Defining Sequences of Methods in Object Interfaces 24
5.2.2 Example of a file Interface 25
5.2.3 Controlling Event Channels with MUSKOX 26
5.3 MUSKOX AS A PARSER TOOL 26

6. ANNOTATED GLOSSARY OF MUSKOX TERMS 27

7. MUSKOX BACKGROUND 38

7.1 MUSKOX AND FORMAL LANGUAGES 38
7.2 MUSKOX AND EVENT-DRIVEN SYSTEMS 39

1. INTRODUCTION

I am the State!
(L'état, c'est moi!) -Louis XIV (1655)

1.1 STATES

MUSKOX is a tool for software engineers to design, program, and maintain software systems using object states. We begin this Concepts Guide by introducing a general concept of state:

1. States in our World
2. Behaviour -- Changing States According to Laws
3. Understanding States
4. States, Computers, and Software Engineering
5. Programming Computers with States Using MUSKOX

1.1.1 States in Our World

The Webster's New Encyclopedic Dictionary lists several meanings for the word *state* including:

'a mode or condition of being'

In this sense we encounter states everywhere, for example:

- Water may be liquid or frozen
- The sky may be blue, grey, or black

1.1.2 Behaviour -- Changing States According to Laws

Usually things change their states, i.e. behave, according to laws. We, as humans, discover the states of things and understand the laws of transitioning between states, for example:

- Liquid water becomes frozen after cooling below a certain temperature
- Blue sky becomes grey when the wind gathers clouds

1.1.3 Understanding States

We also directly and often efficiently use our knowledge of states and laws of state transitions when it is appropriate. For example, understanding the law of transition between the liquid and frozen states of water we may easily produce ice in the refrigerator while preparing for a party. Also, knowing why the sky is grey we may expect rain and, therefore, take an umbrella when going out of the house.

1.1.4 States, Computers, and Software Engineering

It is natural for us to discover the world in terms of states and to act upon a state since we understand the laws of state transitions. However, it is hard for computers to acquire and convert declarative knowledge of states into efficient and reliable executable programs. Therefore, the job of a human software engineer is to transfer that knowledge from his/her head into a computer program.

1.1.5 Programming Computers with States Using MUSKOX

MUSKOX is a tool that helps software engineers to convey their knowledge about the states of things into computers. With MUSKOX, the translation from states into a program does not happen entirely in the programmer's head. Instead, a software engineer explicitly declares objects, object states, and the laws of state transitions in a declarative format of the MUSKOX metalanguage. The MUSKOX compiler then converts the declarations into *efficient* C++ programs. Usually MUSKOX produces programs that run with little or no overhead compared to handcoded programs.

MUSKOX partitions a software system into higher-level design declarations of the laws of state transitions, and a lower-level C++ code that takes care of the details. The MUSKOX compiler converts the declarations of the laws of state transitions into C++ programs that control the application at runtime, thereby making the design a driving part of the running system. *With MUSKOX the higher level design expresses the behaviour of the system in terms of states using state transition rules.*

1.2 INTENDED AUDIENCE AND PURPOSE

The MUSKOX Concepts Guide is intended for software engineers and their managers to:

- Understand the fundamental concepts of object states and state transitions
- Apply these concepts to design, program, and maintain practical software applications

A knowledge of the C++ programming language will help when reading The MUSKOX Concepts Guide, since MUSKOX metalanguage augments C++.

1.3 TYPOGRAPHICAL CONVENTIONS

The following denotes the formatting used throughout this guide:

Format	Usage
Bold type	Keywords, MUSKOX built-in symbols, names of components, headings
Italic type	Major concepts, comments in the examples
`Courier` font	Program fragments and examples

1.4 MUSKOX PRODUCT OVERVIEW

MUSKOX is a Software Engineering Tool composed of three separate pieces:

- The **MUSKOX metalanguage**, for declaring types, objects, object states, state transition rules, communication, and actions that together describe the application's runtime behavior

- The **MUSKOX compiler,** for converting MUSKOX declarations into efficient C++ programs

- The **MUSKOX Application Runtime Monitor,** for controlling current object states and initiating actions based on the states according to the rules of the MUSKOX metalanguage

The MUSKOX compiler compiles declarations into C++ programs. Then a C++ compiler compiles the generated programs into the MUSKOX Application Runtime Monitor (ARM). After that, you debug the ARM with a log of state transitions recorded at runtime, a description of the ARM produced by the MUSKOX compiler, and a C++ debugger.

1.5 APPLICATION DEVELOPMENT WITH MUSKOX

The MUSKOX Software Engineering Tool provides the capability of describing an application's objects during compilation, and of maintaining their correct states at runtime. Application development with MUSKOX proceeds in cycles with five stages:

- Define types, objects, object states, state transition rules, and communication between objects with the MUSKOX metalanguage

- Specify actions, triggers, and reductors -- the C++ code to be executed automatically in the course of object state transitions -- as part of the type definitions

- Convert type and object definitions into C++ programs with the MUSKOX compiler

- Compile and link the resulting programs into the MUSKOX Application Runtime Monitor (ARM) that controls object state transitions and executes actions, triggers, and reductors at runtime

- Run the ARM and debug it using a log of state transitions that the ARM records at runtime, a description of the ARM produced by the MUSKOX compiler, and a C++ debugger of your choice

1.6 MUSKOX DOCUMENTATION

This MUSKOX Concepts Guide is a companion to The MUSKOX Programming Guide and The Muskox Reference Guide. Each guide has a different focus:

- The Concepts Guide explaines MUSKOX concepts, introduces the MUSKOX metalanguage, the MUSKOX compiler, and the MUSKOX run time system. It also talks about applications of the MUSKOX tool.

- The Programming Guide explains uses and applications of MUSKOX in detail. The Programming Guide contains an extensive description of all of MUSKOX's features, supplemented by a tutorial and sample code fragments as practical examples.

- The Reference Guide mainly comprises individual "man pages" of MUSKOX's elements for lookup purposes.

1.7 ORGANIZATION OF THE MUSKOX CONCEPTS GUIDE

This guide, The Concepts Guide, is divided into eight sections:

SECTION 1. **Introduction**, the current section

SECTION 2. **Fundamentals of the MUSKOX Metalanguage**, which describes the concepts of the MUSKOX metalanguage

SECTION 3. **The MUSKOX Compiler**, which describes the MUSKOX compiler and mapping from the MUSKOX metalanguage to the host programming language C++

SECTION 4. **The MUSKOX Application Runtime Monitor (ARM)**, which explains the architecture of the user-extendible Application Runtime Monitor

SECTION 5. **MUSKOX Applications**, which explains how MUSKOX can be used in High Availability Systems, Object-Oriented Frameworks, and Language Environments

SECTION 6. **Annotated Glossary of MUSKOX Terms**

SECTION 7. **MUSKOX Background**, a complementary section that explains the relation between MUSKOX, formal languages, and event-driven systems

2. FUNDAMENTALS OF THE MUSKOX METALANGUAGE

In this section we provide the major concepts of the MUSKOX metalanguage:

1. What is *Programming with States*
2. What are MUSKOX objects and object lifecycles
3. How are states and state transition rules defined and used
4. How do messages cause the state transitions
5. What is the symbolic communication and why is it important
6. How are symbols sent and delivered
7. What is the MUSKOX State Transition Engine and how does it operate
8. When are actions executed during the lifecycle
9. How do different objects execute their lifecycles concurrently
10. What is the MUSKOX object model

At the end we also compare traditional programs and MUSKOX lifecycles. The MUSKOX Programming Guide and The MUSKOX Reference Guide contain more detailed explanations of the elements of the MUSKOX metalanguage.

2.1 PROGRAMMING WITH STATES

2.1.1 States in Traditional Programming

Viewed from an object-oriented perspective a computer program consists of *objects* and their associated *states*. *The history of state transitions for a particular object constitutes that object's lifecycle. Together, lifecycles of many objects represent the behaviour of the program.*

In traditional programming a state is vaguely understood as the instantaneous contents of the variables of a program or the attributes of an object, but has no formal meaning. This precludes description of object lifecycles due to the enormous number of possible states.

Historically programmers encoded a limited number of states in enumerations and bit-fields to circumvent the problem, and build the logic of their programs around the states with *switch-case, if-then-else*, and other procedural control flow statements. Programs that run efficiently on modern computers are created in this way.

In this approach a programmer translates a declarative design of the application into the procedural logic. Sometimes a design tool may help generate the program code, but often the translation takes place primarily in the programmer's head. Subsequent maintenance is often difficult and costly because the resulting program does not express the concepts of the design directly. *Even if the data are represented structurely and with objects, the behaviour of such a program is burried in a cloud of details.*

2.1.2 Declaring States with the MUSKOX Metalanguage

In MUSKOX there is no mismatch between design concepts and efficient programming. *You build an application from objects with explicitly declared* states *and* order of state transitions. Design concepts are expressed directly using MUSKOX elements, with lower level details of the application encoded in C++.

The MUSKOX compiler translates state transition rules into C++ numeric arrays and converts C++ code into the object's methods. You compile the generated code with a C++ compiler into the MUSKOX Application Runtime Monitor (ARM). *The ARM is the executable portion of the application*. At runtime the ARM controls state transitions with arrays with little or no overhead. The ARM executes your C++ code during state transitions by calling generated methods. The ARM also produces a log of state transitions for you or the ARM to analyze on- or offline.

The art of using MUSKOX to produce an efficient Application Runtime Monitor is to capture the most important application states and encode the behavior of the application with state transition rules, and leave lower-level details for C++ programs.

2.2 OBJECTS AND OBJECT LIFECYCLES

Every MUSKOX object is a part of the ARM and an instance of a MUSKOX type. It is important to understand that states and messages are properties of objects. An object undergoes its *lifecycle* by:

- Transitioning through states
- Exchanging messages with objects, and receiving messages from outside of the ARM
- Executing C++ code during state transitions

The MUSKOX compiler converts states and messages into nonnegative numbers. MUSKOX runtime uses these numbers for state transitions and communication.

2.3 STATES AND STATE TRANSITION RULES

Each object may be in a state at a given point in time. A state is a MUSKOX element declared with a **state** keyword in a type definition. For example:

```
state open closed ioDone;
```

Object states are defined in order to write state transition rules. A rule simply declares a sequence of states that leads to some target state. For example, consider the following rule:

```
jobDone : open ioDone closed;
```

The rule says that states `open`, `ioDone`, and `closed` is a valid sequence of states for the object. The object concludes in a target state `jobDone` after the transition from `open` to `ioDone` to `closed`.

A rule declares the order of transitions from state to state. For example, the previous rule does not allow the sequence of states

```
open closed ioDone
```

Note that the MUSKOX metalanguage is position-independent -- that is spaces, tabs, and new line symbols may be used freely for formatting. For example, the previous state transition rule may be equivalently written as

```
jobDone :     open
              ioDone
              closed;
```

For a complete explanation of lexical elements please consult The MUSKOX Reference Guide.

2.3.1 Combining State Transition Rules

The key feature of state transition rules is that the target state on the lefthand side of a rule may be a state in the righthand side sequence of the same or some other rule. For example, the above target state jobDone may appear in the righthand side of another rule:

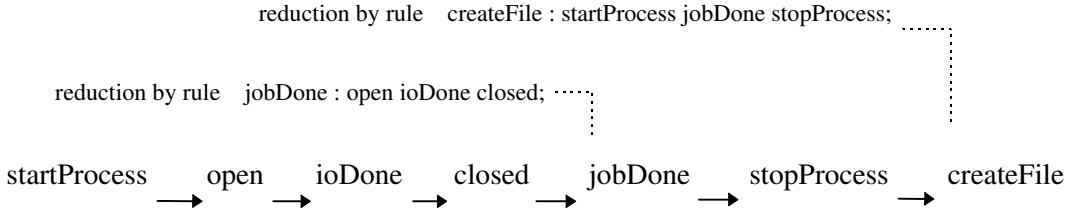
```
createFile : startProcess jobDone stopProcess;
```

The semantics of state transition say that an object goes from one state to another in an order specified in a rule. *When an object transitions through the states that constitute the righthand side of a rule, it results in the target state specified on the lefthand side of the rule. The process then repeats for the same or other rule.*

The final state transition to the target state is called a *reduction* of the rule. For example, when combined the two preceding rules allow an object to pass through the following sequence of states:

```
startProcess
open
ioDone
closed
jobDone
stopProcess
createFile
```

The following graph illustrates the sequence of state transitions and reductions:



2.4 MESSAGES

An object state change is usually initiated when an object receives a message, either from itself, from another object, or from outside of the ARM. There are special communication instructions for sending messages between objects.

In MUSKOX, messages are declared with the **message** keyword, for example:

```
message Open Close Read Write;
```

Messages may also appear in a state transition rule, but only in the righthand side. *States and messages declare a valid sequence of state transitions and received messages, that result in the object entering the target state.* For example, the rule

```
jobDone : open Write closed;
```

declares that while an object is in the state `open` , it receives a message `Write`, then transitions into the state `closed`, and after that transitions into a target state `jobDone`.

With messages we can create a more complete set of rules to deal with files, for example:

```
jobDone : open ioDone closed;
open    : Open;
closed  : Close;
ioDone  : <Read | Write>+;
```

The first rule has three states in the right-hand side. The second and the third rules both have a single message in the righthand sides. The second rule declares that after receiving a message `Open` an object transitions into the target state `open`. The third rule declares that after receiving a message `Close` an object transitions into the target state `closed`. The last, fourth rule, declares that after receiving a series of messages `Read` and/or `Write` an object transitions into the target state `ioDone`.

The vertical bar above is shorthand for alternative righthand sides; the items enclosed in `'<' '>'` and followed by `'+'` constitute a *regular expression* that denotes a repetition of any of the alternatives (`Read` o r `Write`) . The last rule with the regular expression subsumes the following rules without the regular expression:

```
ioDone  : Read;
ioDone  : Write;
ioDone  : ioDone Read;
ioDone  : ioDone Write;
```

The first two rules declare that an object transitions into the target state `ioDone` after receiving a message `Read` or `Write`. The last two rules declare that while in the state `ioDone` an object may receive a message `Read` or a message `Write` -- in either case it transitions into the same state `ioDone.`

2.5 SYMBOLIC COMMUNICATION

As you already know from subsection 2.2, the MUSKOX compiler converts definitions of states and messages into nonnegative numbers. At runtime messages and states are represented by these numbers: when an object receives a message it actually receives a numeric code generated for the message by the MUSKOX compiler. *The important feature of the communication in MUSKOX is that an object may send and receive not only a numeric code of a message, but also the numeric code of a state.*

That is, MUSKOX objects communicate with both states and messages. Receiving a state is equivalent to transitioning into the state following a reduction of any rule for which the state is a target. *This equivalence of states and messages is important for runtime debugging and rapid prototyping, since objects transition directly to a target state when receiving it, instead of undergoing a complete lifecycle declared in the righthand side of a rule.* For example, with the rule

```
ioDone : <Read | Write>+;
```

a transition to the state ioDone may be achieved by receiving either

- a series of messages `Read` or `Write` (complete lifecycle), or
- just the state ioDone

Note that in MUSKOX the term *symbol* denotes both state and message, due to the equivalence between states and messages.

2.6 NON-BLOCKING QUEUEING MECHANISM

MUSKOX sends and delivers symbols, states and messages, at runtime with its non-blocking queueing mechanism: *An object continues its lifecycle after sending a symbol without awaiting the symbol's delivery to its destination* . Each object has a queue through which it receives symbols from other objects, from itself, and from outside of the ARM.

2.7 THE STATE TRANSITION ENGINE

The runtime State Transition Engine (engine for short) supplied with MUSKOX is a part of the ARM and controls lifecycles of all the objects in the system. Based on the rules of the MUSKOX metalanguage, the engine checks the correctness of the sequence of transitioned states and received symbols, and performs state transitions and reductions. During its work the engine utilizes the numeric arrays prepared by the MUSKOX compiler based on the definitions of state transition rules.

For the engine to control the lifecycles, each MUSKOX object contains a stack of symbols where it accumulates already transitioned states and received messages. The following steps explain how the engine controls object lifecycles with the stacks:

- When a symbol is received the engine shifts it on the stack. *Shifting a symbol which is a state is a state transition*.
- Then the engine looks for a state transition rule whose righthand side constitutes the top of the stack.
- A reduction occurs if such a rule exists: The stack is reduced by the number of elements on the righthand side of the rule, and the target state is shifted on the stack. *This shift is a state transition.* After the shift the engine looks for a matching rule to reduce the stack again, and repeats this step while such a rule exists.
- If there is no adequate rule to reduce the stack, the engine waits for a symbol to arrive to the object, or switches to another object if a symbol has arrived to the other object.
- When the engine reaches a point where no further reduction is possible, or where a received symbol cannot be shifted since there will be no rule to reduce the stack with this symbol in it, then the engine usually either reports that the final state has been reached or diagnoses an error.

For example, given a set of rules:

```
createFile : startProcess jobDone stopProcess; // rule 1[1]
jobDone    : open ioDone closed;               // rule 2
open       : Open;                             // rule 3
closed     : Close;                            // rule 4
ioDone     : Read;                             // rule 5
ioDone     : Write;                            // rule 6
ioDone     : ioDone Read;                      // rule 7
ioDone     : ioDone Write;                     // rule 8
```

the following sequence of states and messages

```
startProcess Open Write Read Close stopProcess
```

causes the engine to perform the following chain of shifts and reductions:

Arriving Symbol	**Engine Steps**	**Stack Content**
		bottom → top
`startProcess`	shifts	`startProcess`
`Open`	shifts	`startProcess Open`
	reduces rule 3	`startProcess open`
`Write`	shifts	`startProcess open Write`
	reduces rule 6	`startProcess open ioDone`
`Read`	shifts	`startProcess open ioDone Read`
	reduces rule 7	`startProcess open ioDone`
`Close`	shifts	`startProcess open ioDone Close`
	reduces rule 4	`startProcess open ioDone closed`
	reduces rule 2	`startProcess jobDone`
`stopProcess`	shifts	`startProcess jobDone stopProcess`
	reduces rule 1	`createFile`

[1] The text after // in MUSKOX specifications denotes comments.

2.8 ACTIONS

When a MUSKOX object transitions into a state, a C++ program called *action* may be executed by the ARM. Actions are defined within the state transition rules: An object shifts on the stack a symbol specified before the action, executes the action, and then it is ready for shifting the next symbol. The object undergoes a reduction after the action if the action is the last item in the rule.

An action may contain any valid C++ code. The following are examples of actions:

```
open    : Open { fid = fopen("file", "r"); };
closed  : Close { fclose(fid); };
jobdone : open { cerr << "\nFile is open"; } ioDone close;
```

The importance of actions is hard to overestimate. *All of the work in the ARM is done with actions, because states and messages merely serve as a logical skeleton to declare the order of execution of different actions from the same object, and to synchronize actions executed in parallel by different objects.*

MUSKOX metalanguage also allows *triggers* and *reductors* -- those are C++ programs similar to actions associated with states and state transition rules that the engine executes in the course of state transitions and communication.

2.9 CONCURRENCY

Every MUSKOX object has its own independent thread of execution. *Each object undergoes its lifecycle (i.e. changes its states, receives and sends symbols, and executes actions) in sequence. However, all of the objects may perform their lifecycles in parallel with each other.* Consequently, the object and its lifecycle are the fundamental unit of concurrency.

2.10 THE MUSKOX OBJECT MODEL

States and state transition rules are properties of MUSKOX objects which are instances of MUSKOX types and are part of the ARM. In this subsection we provide more details on MUSKOX types and objects:

1. The MUSKOX Type
2. The MUSKOX Object
3. Object Lifetime
4. Links and Communication
5. Synchronization of Lifecycles

2.10.1 The MUSKOX Type

A MUSKOX type is a C++ class augmented by a definition of MUSKOX elements: States, messages, state transition rules, actions, triggers, reductors, and others. A hierarchy of types may be defined by inheritance in the same way as a hierarchy of classes in C++. A derived type inherits and redefines data and member functions from its base types and classes according to the semantics of C++. Also, MUSKOX elements from the base types may be either inherited or redefined in the derived type according to MUSKOX semantics.

2.10.2 The MUSKOX Object

A MUSKOX object is an instance of a MUSKOX type and is a part of the ARM. Each MUSKOX object implicitly inherits from a supplied C++ virtual base class `typeProto`. The class contains a number of useful data and member-functions, such as:

Lifecycle Members

- Query the current object state and recent history of state transitions
- Determine the states of the object's children (children are objects connected to a parent object through a link declared in the type of the parent object)
- Block and unblock an object from communication with other objects
- Repost delayed symbols (i.e. symbols that had arrived unexpectedly and therefore were delayed until the appropriate repost time)

Framework Methods

- Link and unlink parent and children objects
- Verify links between objects
- Create, clone, and delete dynamic objects
- Verify if an object is static or dynamic

Communication Methods

- Send states and messages

2.10.3 Object Lifetime

MUSKOX objects are created either statically when compiling or dynamically at runtime. Each statically defined object has a C++ name that is subject to C++ scoping rules. The name may therefore be used in the same manner as any C++ name.

Dynamic MUSKOX objects are created and deleted at runtime. Names of dynamic objects do not belong to the C++ namespaces.

Names of static and dynamic objects are used by the Framework Methods from the `typeProto` class to access objects within the ARM.

2.10.4 Links and Communication

A type may specify a number of links. A link is a named handle through which an object, called *parent*, exchanges symbols, i.e. states and messages, with another object, called *child*. Parent and children objects synchronize their state changes by sending and receiving symbols along the links.

A type may define a number of links to be established for any of its instances at compile or runtime. Chains of links between different objects may form arbitrary graphs -- trees, cycles, etc.

2.10.5 Synchronization of Lifecycles

Each running object transitions from state to state following a single thread of execution. However, the lifecycles of objects are independent of each other, and the ARM may execute them in parallel.

MUSKOX synchronizes lifecycles of individual objects in two ways, by:

- Exchanging symbols, i.e. states and messages between objects. Because objects expecting symbols will suspend execution of their lifecycles, declarative state transition rules explicitly declare the order in which an object effects transitions from state to state and the order of exchanging symbols with itself or other objects.

- Automatically notifying objects of state changes. A rule that involves states of other objects allows a state transition for a particular object only after those other objects transition into specified states.

2.11 PROGRAMS AND LIFECYCLES

This section concludes the fundamentals of the MUSKOX metalanguage comparing procedural programs and MUSKOX lifecycles.

2.11.1 Differences

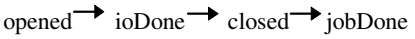
Usually program statements have no names (unless labeled), while states and messages are always named. *The advantage of lifecycles is that the rules directly express application behavior with named communication and messages. At runtime the ARM maintains the current state and produces a history of state transitions in a log.* Tools and programmers have access to the current state, and may analyze a history of communication and state transitions from the log and object stacks.

2.11.2 Similarities

According to the principles of structured programming programs may be built from statements grouped in three basic patterns: Sequences of statements, repetition (*e.g. do-while* loops), and selection (e.g. *if-else* conditions). Lifecycles also may be described by these patterns:

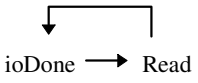
Sequence Occurs when states are transitioned and symbols received following a righthand side of a rule, culminating in the target state. For example:

```
jobDone : opened ioDone closed;
```



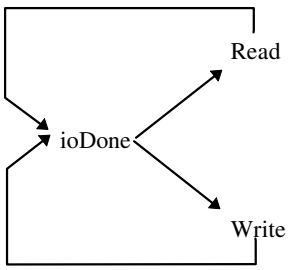
Repetition Occurs when the target state of a rule appears in the righthand side of the same rule. For example:

```
ioDone : ioDone Read;
```



Selection Occurs when a symbol is followed by different symbols in two or more rules. For example:

```
ioDone : ioDone Read
ioDone : ioDone Write;
```



2.12 CONCLUSION

We have now concluded the fundamentals of the MUSKOX metalanguage -- the first of the three components of the MUSKOX Engineering Tool. Please refer to Sections 3 and 4 for information on the other two components -- the MUSKOX compiler and the MUSKOX Application Runtime Monitor.

3. THE MUSKOX COMPILER

In this section we explain how the MUSKOX compiler converts declarations of types, objects, states, and other MUSKOX elements into the C++ programs. The plan of the section is:

1. MUSKOX Specs
2. Files Generated by the MUSKOX Compiler
3. MUSKOX and the Host Programming Language C++
4. Mapping to the Host Language C++

3.1 MUSKOX SPECS

A MUSKOX *spec* is a logical sequence of elements from the MUSKOX metalanguage. The MUSKOX compiler accepts MUSKOX specs prepared in source files with an editor, or by other means. The files with the specs must have the extension `'.ms'`.

3.2 FILES GENERATED BY THE MUSKOX COMPILER

The compiler converts files with specs into C++ source files, and also creates a number of additional files. For a spec `file.ms` the MUSKOX compiler creates files with the name `file`, but with different extensions listed in the following Table 1:

Extension	Description	Examples
`.h`	Contains C++ class definitions generated by the MUSKOX compiler from corresponding MUSKOX types defined in the `.ms` file.	`t5152.h`

`.C` or `.cpp`	The main file generated by the MUSKOX compiler. It contains:
	• Methods that are images of the original actions, triggers, and reductors
	• Numeric arrays with parsing tables resulting from processing state transition rules
	• Definitions of static C++ objects that are images of MUSKOX static objects.
	This file automatically includes the generated `.h` file.
		`t5152.C`
		or
		`t5152.cpp`

`.m`	Contains **`#define`** statements for numeric values of states and messages. The file may be included in any C or C++ files that do not include generated `.h` files, but may still contain code that communicates these states and messages to MUSKOX objects.	`sym.m`
`.v`	Contains a self-explanatory description of the ARM.	`t5152.v`
`.d`	Contains a hierarchy of attached spec files in a format similar to the dependency list of the UNIX make utility.	`t5152.d`

Table 1. File Extensions Created by the MUSKOX Compiler

3.3 MUSKOX AND THE HOST PROGRAMMING LANGUAGE C++

MUSKOX is always used with a fullfledged *host programming language*. The current version is implemented as a preprocessor to the host language C++. The MUSKOX metalanguage and the host language divide their functions in the following manner:

- The MUSKOX metalanguage controls processes by defining object lifecycles via object states, state transition rules, and interobject communication.
- The host language C++ encodes MUSKOX actions, triggers, and reductors: these are methods executed in the course of state transitions.

3.4 MAPPING TO THE HOST LANGUAGE C++

The MUSKOX compiler maps elements of the MUSKOX metalanguage into C++ elements. Some of the converted elements may be accessible from user-written C++ code, but some are not accessible -- the ARM uses them transparently from the user. The following table lists the conversions between MUSKOX elements and C++:

MUSKOX Elements	Accessible from C++	C++ Elements
MUSKOX type	Yes	C++ class with a virtual base class `typeProto`.
C++ data members	Yes	C++ data members remain intact.
C++ member-functions	Yes	C++ member-functions remain intact.
MUSKOX symbols -- states and messages	Yes	A table of symbol values containing character symbol names and generated numeric values. Depending on the MUSKOX compiler command line flags, public symbol names are converted into one of the following representations: • Calls to the generated virtual functions that return the numeric values of symbols. • Macro definitions that result in searching a table of symbol values at run-time based on the character representation of the symbol.
State Transition Rules	No	Numeric arrays with parsing tables. The engine uses these arrays to shift symbols on the stack and to apply stack reductions according to the state transition rules.
Action	No	C++ member-function.
Trigger	No	C++ member-function.

Reductor	No	C++ member-function.
Communication instruction	No	C++ member-function that invokes a communication method from virtual base class `typeProto`.
Extracode	Yes	Moved into C++ code intact.
Hierarchy of MUSKOX types and C++ classes	Yes	C++ class hierarchy.
Link	Yes	C++ classes, C++ objects, and member-functions.
Static MUSKOX object	Yes	Static C++ object.

Table 2. MUSKOX to C++ Conversion

4. THE MUSKOX APPLICATION RUNTIME MONITOR (ARM)

The ARM executes application logic by processing the lifecycles of its objects. We describe the ARM in the following subsections:

1. Building the ARM
2. The Framework
3. The State Transition Engine

4.1 BUILDING THE ARM

The MUSKOX compiler converts declarations of types and objects from MUSKOX specs into C++ programs. Then the C++ compiler compiles and links the resulting programs with the MUSKOX framework, MUSKOX State Transition Engine, and other software into the MUSKOX *Application Runtime Monitor* (ARM). The following figure graphically illustrates how to build the ARM:

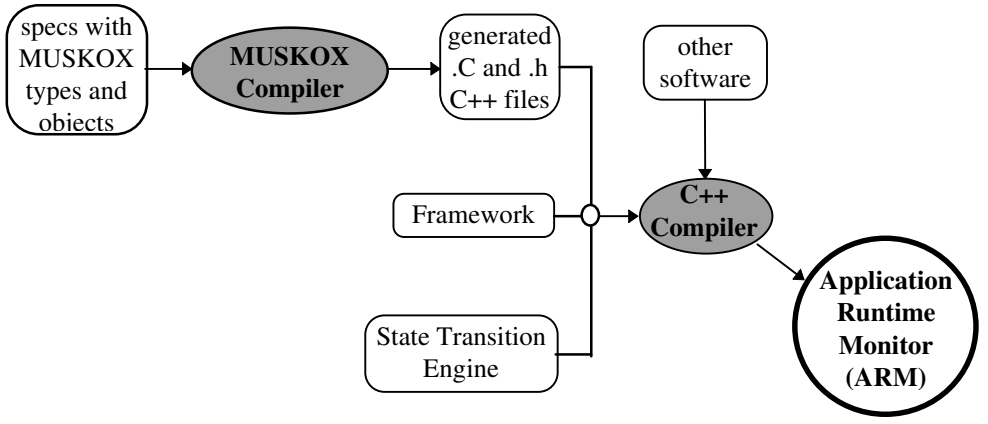


Figure 1. Building the ARM

4.2 THE FRAMEWORK

The *framework* is a part of the ARM. It maintains the runtime type information, numeric arrays with parsing tables, object name-space, object links, and queues of symbols (states and messages). The framework also:

- Initiates the ARM during startup
- Dispatches symbols to objects
- Invokes a State Transition Engine to accept or reject the symbols
- Logs the stream of symbols in the log file

Developers may either use the *standard framework* that is supplied with MUSKOX, enhance the standard framework, or write their own framework.

4.2.1 Standard Framework

The *standard framework* is a framework supplied with MUSKOX. It maintains object state changes and communication within the ARM. The standard framework provides communication between objects via a local queue. Each symbol in the queue contains the virtual memory addresses of both source and destination objects.

Standard framework does not support communication between MUSKOX objects from different ARMs. However, developers have the option to provide communication from ARM to ARM and global object name services by modifying the source code of the standard framework and the `typeProto` class.

4.3 THE STATE TRANSITION ENGINE

The part of the ARM called the *State Transition Engine* is the core of the MUSKOX runtime system. The engine receives states and messages for every object and accumulates them on the *parsing stack*. It then selects, if possible, a next rule to reduce the stack by matching the rule with elements from the top of the stack. As a result of stack reduction an object transitions to a target state prescribed by the rule. The target state is also shifted on the stack. The engine logs state transitions and received symbols in a MUSKOX log file.

The standard framework calls the engine when it extracts a symbol from a local queue. The engine executes a state transition algorithm for the object to which the symbol is addressed.

The engine uses parsing tables generated by the MUSKOX compiler. It executes user-written C++ code -- actions, triggers, and reductors -- in the course of state transitions. The code implements the logic of the application and communicates states and messages between objects.

5. MUSKOX APPLICATIONS

The preceding sections covered the MUSKOX fundamentals. This section describes some specific applications and classes of applications where MUSKOX is especially suitable to build complex systems with concurrent and distributed components:

1. A High Availability-System with the kernel built from the MUSKOX ARM
2. An application of MUSKOX technology to Object-Oriented Frameworks
3. MUSKOX as a parser tool to process formal languages

5.1 HIGH-AVAILABILITY SYSTEMS

In general, a High-Availability System maintains a number of software and hardware network resources for users of the resources. In the event of one or more resources failing the system provides a fast switchover to an alternative resource.

Examples of resources:

- Network connections
- File systems and disks
- Power
- Hosts
- User applications
- Database servers, WEB servers, and Transaction Monitors

Examples of failures:

- Network breakdown
- Loss of power
- Application core dumps
- OS panics
- Host shutdown
- Disk data corruption

The technical challenge in the design of a High-Availability System is to control *complex* behavior of a *huge* number of resources and provide *fast* switchover. A medium-size High-Availability system may have

1. Tens and more types of resources
2. Tens and more states, and hundreds of state transitions per type of resource
3. Hundreds and thousands resources of different types at runtime

Failure of critical resources may require execution of hundreds and thousands of OS scripts in parallel to provide alternative resources within seconds or within a few minutes. To provide High Availability the system must:

Before runtime create a model of behavior for resources by answering the questions:

- What are the types of resources?
- What are the states of resources and possible state transitions for each type of resource?
- How do state transitions of some resources depend on the state transitions of other resources?
- What causes state transitions in the first place: e.g. scripts, operator commands or malfunctioning?

Based on the model, create custom configurations for a particular network cluster:

- Describe each resource
- Determine resources whose states depend on each other
- Define scripts that will force resource state transitions

At runtime:

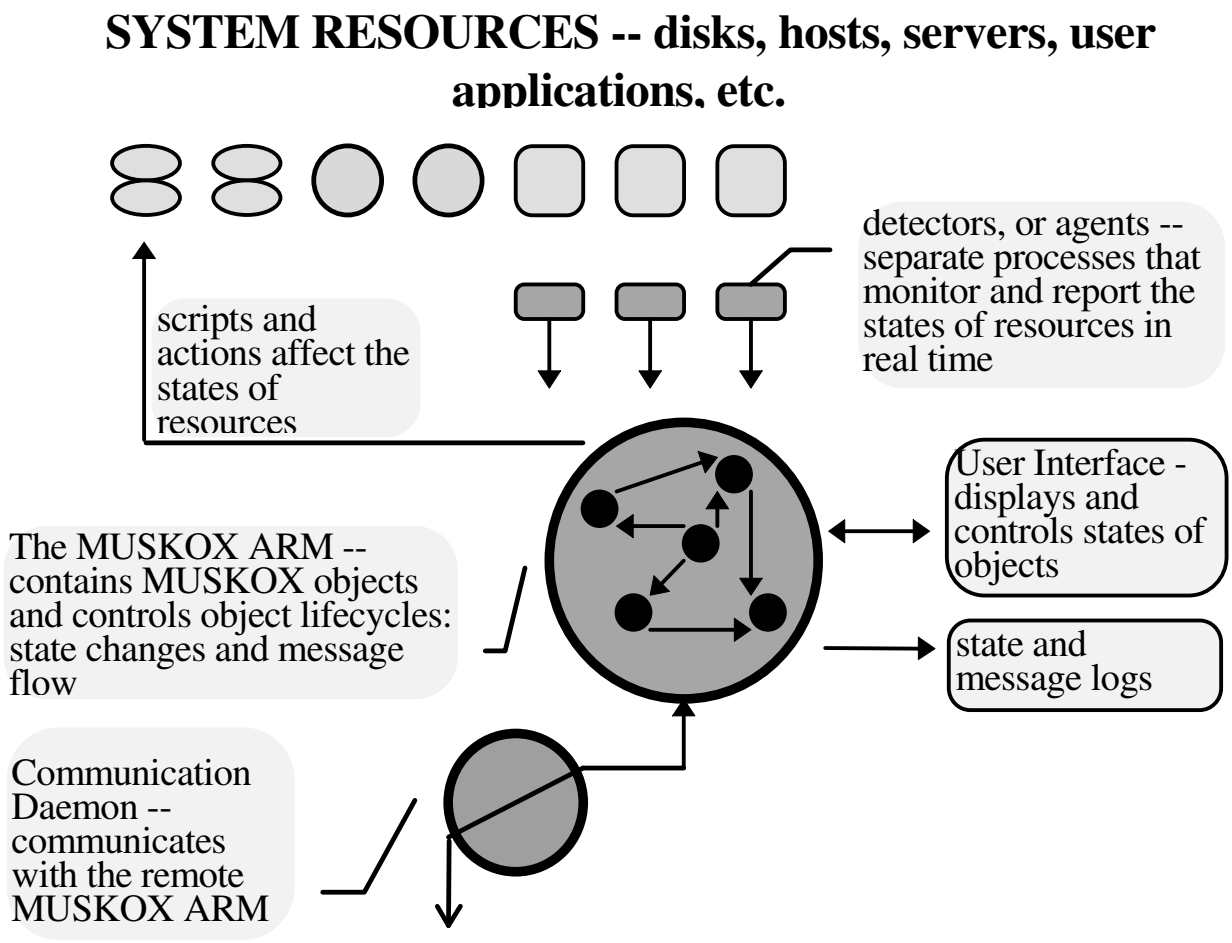
- Keep track of current resource states and make the states observable with a user interface
- Provide a log of state transitions for every resource
- React to failures by running in parallel scripts that make alternative resources available

5.1.1 MUSKOX and High-Availability Systems

A High-Availability System is inherently complex: it must support a large number of resources and control their state changes in real time in an orderly fassion. The MUSKOX Software Engineering Tool manages the complexity of network configurations by describing hardware and software resources as objects which are instances of MUSKOX types. Each type of resource describes the rules of state transitions and possible states through which a resource transitions during its lifetime.

A type hierarchy reuses the code and the state transition rules that control common behaviour of resources of different types. Links between resources define dependencies between the state changes of different resources. Actions executed by the MUSKOX ARM in the course of state transitions at runtime control startup and shutdown operations, and provide a switchover when a critical resource fails. MUSKOX may provide High Availability for large configurations of resources on small and big hosts.

The ARM build for a High Availability System is part of the architecture described below:



The detectors (agents), User Interface, Communication Daemon and the ARM are all separate processes that operate synchronously and communicate via the OS or network mechanisms. Detectors report the states of system resources in real time by sending messages to the ARM when they detect a state change. Consequently, there is little or no message flow in the system if the system is stable .

The ARM contains MUSKOX objects -- each system resource has a corresponding MUSKOX object in the ARM. The ARM directs a message from a detector about a state change of a particular resource to the correspondng MUSKOX object and applies that object's state transition rules. Then an action may be executed. An action may fork a script that will be executed in parallel with other scripts and components of the system. Actions may run scripts to mount, unmount, configure and reconfigure disks, start up or shut down applications, databases, or WEB servers, etc. System resources will be affected by these actions and change their states as a result. Detectors report the state change back to the ARM, and the whole process continues like a loop of resource state changes and reactions from the ARM.

The Communication Daemon may exchange messages with the ARM resident on another host: remote hosts keep track of each other's states and start up local applications when they detect that a remote host has failed.

5.1.2 Advantages of Using MUSKOX in High-Availability Systems

The key advantages of using MUSKOX in the High-Availability System are the following:

- Extracting the logic of resource state transitions out from detectors, user interfaces, and communication mechanisms separates engineering tasks for both development and maintenance of the system: Most of the engineers concentrate on specific issues of detecting and reporting the current states of particular resources, and on user interfaces. At the same time the higher level logic of inter-resource dependencies, and state transition logic of the particular types of resources are designed, coded, and maintained only by those responsible for the MUSKOX ARM.

- The high level of concurrency of forked scripts during execution of MUSKOX actions allows state change of a huge number of resources in a controlled manner within a short period of time during startup, shutdown, and switchover. This is the main service provided by a High-Availability System.

5.2 MUSKOX AND OBJECT-ORIENTED FRAMEWORKS

5.2.1 Defining Sequences of Methods in Object Interfaces

Currently a number of Object-Oriented Frameworks are commercially available for building complex distributed applications. These frameworks usually permit definition of types and objects that are instances of the types. As a part of the type definition developers provide a set of object methods that constitute the type interface to the outside world. While the framework provides a way to specify each individual method, the logic of calling the methods is usually not supported by the framework mechanism itself. Instead, a developer encodes that logic with a programming language at hand.

MUSKOX can remedy this deficiency in Object-Oriented Frameworks. By augmenting type definitions with object states, and by associating methods from the type interface with actions executed during state changes, developers may clearly define a valid sequence of invocations of type methods. By making object states observable by other objects, tools, and programmers, developers may increase the quality and speed up application development and maintenance.

5.2.2 Example of a file Interface

Consider a basic example of a type `file` in a hypothetical Object-Oriented Framework. This type may provide methods to open, read, write, and close the file. The framework itself does not provide any means to express the rule that before reading or writing the file must be opened, and that file must be closed after I/O is finished.

To remedy this MUSKOX defines states `open`, `io`, and `closed` in the type `file`, and provides rules for transition between these states. State transitions are triggered by the corresponding messages `Open`, `Read`, `Write`, and `Close`. Actions and triggers associated with the messages invoke the methods that do the actual job, while state transition rules control the sequence of application of the methods. The following is a simplified example of the type `file`:

```
type file
{
public:                  // methods -- procedural interface
      int fOpen(...);
      int fClose(...);
      int fWrite(...);
      int fRead(...);
}                        // state transitions -- declarative interface
[
      state   public open closed io;
      message public Open Read Write Close;

      trigger Read  { fRead(...); }       // execute methods
              Write { fWrite(...); };     // when a suitable
                                          // message arrives

      LIFE : open io close;               // the state transition
                                          // rules define
      open  : Open  { fOpen(...); };      // the valid sequence of
      close : Close { fClose(...); };     // received messages,
                                          // transitioned states,
      io : <Read | Write>                 // and executed methods
         | io <Read | Write>;
];
```

5.2.3 Controlling Event Channels with MUSKOX

When an Object-Oriented Framework provides an event channel that communicates events between objects, the sequence of events that travels through these channels is controlled by the procedural logic of the application. Yet this logic is easier to express, code, and maintain with declarative constructs similar to the logic of state machines.

A simplest example is an event loop in graphical environments -- the loop accepts events generated by the movement of the mouse and by pressing buttons on the keyboard or screen. Usually events are dispatched within the loop with *switch/case* and *if/then/else* statements. The dispatch may be programmed with the MUSKOX metalanguage by defining valid sequences of events and states of objects (windows, buttons, etc.) to which events are directed. MUSKOX is advantageous since it makes the event logic explicit in the system, and allows reuse of the logic through MUSKOX inheritance of states, messages, and state transition rules.

5.3 MUSKOX AS A PARSER TOOL

The MUSKOX compiler may be used as a full LR1 parser generator, and the MUSKOX engine as a skeleton parsing engine for formal languages. This is possible since MUSKOX states are equivalent to nonterminals, MUSKOX messages to terminals, and MUSKOX state transition rules are equivalent to context-free rules from context-free grammars. Inheritance, separate stack objects, user-defined lexical analysis, and user-defined error recovery allow the building of multiple recursive parsers within the same executable. In most cases the speed of the engine is comparable to, or faster than, engines from other commercial parsers.

The MUSKOX state transition engine used outside of the MUSKOX Framework is called a *standalone parser*. An example of a standalone parser is the MUSKOX compiler itself : a grammar for the MUSKOX metalanguage is written in the MUSKOX metalanguage, so that the MUSKOX compiler is bootstrapped by itself. MUSKOX-generated standalone parsers may compile and execute languages from within and outside of the ARMs.

6. ANNOTATED GLOSSARY OF MUSKOX TERMS

MUSKOX Element	Explanation

abstract type	A type for which no objects can be created. Used to define types of links and base types. Declared with the abstract directive.
	`abstract type idea[];  // abstract type`
	`type plan(idea fix)[];  // non-abstract type with a link`

action	C++ code on the righthand side of a rule. Action is executed after symbols to its left are shifted on the parsing stack. `LIFE : Hello { cerr << "\nThis is an action"; };`
action conflict	A situation when a type has two or more state transition rules with actions that have same targets and the same sequence of symbols in the right handsides of the rules. Action conflicts are diagnosed by the MUSKOX compiler as errors.
ambiguity	A situation when more than one reduction, and/or a reduction and a shift are possible for the engine to undertake. Ambiguities are resolved with the MUSKOX qualifiers **left**, **right**, **nonassoc**, and **prec**.
Application Runtime Monitor, ARM	The executable portion of the application, built from the MUSKOX specs by first compiling with the MUSKOX compiler, then with a C++ compiler.
ARM	See Application Runtime Monitor.
attribute	Data transfered with a symbol between and within objects during communication and state transitions. Attributes can be accessed from actions, triggers, and reductors. `LIFE : Hello { cerr << "\nThis is an attribute: " << $1; };`
attribute class	C++ built-in type, struct, union, class, typedef, or C++ type expression that defines the shape of an attribute. `attribute {char *} message Hello;`
base list	List of MUSKOX types and C++ classes from which a derived type inherits its elements. `type derived : base1, base2[];`
blocked object	An object that does not receive symbols sent by other objects. Object becomes blocked after invoking `Block()` method, and unblocked after invoking `UnBlock()`. After `UnBlock()` the object receives symbols accumulated on the symbol queue.

built-in symbol

System-defined symbols with defined name and semantics. The built-in symbols are:

```
ACCEPT    ENDoFsTREAM    ERROR      DUMMY     LIFE
No        OTHER          REDUCTOR   TRIGGER   Yes
```

child

Object connected to a parent object by a link.

```
type child[];
type parent(child link)[];
child c;                    // this is a child object
parent(c) p;                // this is a parent object
```

command line flags

Options specified on the MUSKOX compiler command line to control its work and the way it generates the C++ programs.

comment

Text within MUSKOX specs that explains the meaning of the specified MUSKOX elements. Comments do not affect execution of the ARM.

communication

Mechanism for sending symbols between or within objects by communication methods and communication instructions.

communication instruction

Special instruction in triggers, reductors and the righthand side of the rules to communicate a symbol between objects. The MUSKOX compiler translates communication instructions into actions that invoke a communication method.

```
LIFE : Hello (->);        // propagate Hello to children
```

communication method

Method from base class `typeProto` that sends symbols to objects.

```
LIFE : Hello { SendChildren(Hello); }; //use comm. instr.
```

concurrency

Parallel execution of object lifecycles for multiple objects.

construction phase

A period of execution during the startup of the ARM that precedes execution of the `main()` function of the ARM.

C++ code

Compound C++ statement to define actions, triggers, reductors, attribute classes, and extracode.

```
trigger Hello {cerr << "\nThis is a C++ code";};
```

default attribute class

Attribute class assigned by default to the attribute of a symbol if no attribute class is specified for the symbol. It is defined with the **attribute** keyword.

```
attribute int;
```

delay queue

Queue where the engine stores delayed symbols that have arrived unexpectedly. Every object has its own delay queue. The delay queue may be reposted by the `typeProto::RePost()`.

delayed symbol

Symbol defined with the **delay** qualifier. Delayed symbols are postponed in a delayed queue when they arrive unexpectedly until the appropriate repost time:

```
state delay startTransaction;
```

description of the ARM

Description of an LR1 automaton that the MUSKOX compiler generates in a `.v` file.

dynamic object

Object created at runtime with a `Clone()` method from the `typeProto` class, or a method `t::Create()` generated by the MUSKOX compiler for each non-abstract type `t`:

```
newStar = star->Copy();
car333  = carType::Create();
```

error symbol

The error symbol specified in the error directive. The engine sends error symbol to objects if a parsing error occurs and if the symbol is expected.

```
error oops;
```

engine

See State Transition Engine

extracode

C++ code in curly brackets outside of a rule, trigger, and reductor. Moved without change into generated C++ programs.

```
{
#include <iostream.h>                    // extracode
}
LIFE : Hello { cerr << "This is an action";};
```

framework

Part of the ARM responsible for runtime type information, object name space, dynamic objects, object links, and symbol queues. Also dispatches symbols to objects, invokes the engine, and logs the symbol stream.

framework methods

Public methods from class `typeProto` that create, copy, delete, link, unlink objects, and verify object links.

```
newStar->Delete();
```

frozen object

An object that does not accept symbols neither from another objects, nor from itself. The object becomes frozen after calling `Freeze()` , and unfrozen after calling `UnFreeze()` . After `UnFreeze()` it resumes to receives accumulated on the symbol queue.

grammar

Type grammar.

grammar inheritance

The way to build type grammars for derived types: Union of type grammars from base types with the rules, triggers, and reductors from a derived type.

```
type cold[state public hi bye; randevue : hi bye;];
type hot[state public hi kiss bye;
         randevue : hi kiss bye;];
type meet : cold, hot[]; // has both rules
                         //     randevue : hi bye;, and
                         //     randevue : hi kiss bye;
```

host language mapping

MUSKOX compiler process to map MUSKOX types and elements into C++ classes and elements.

inheritance

Reuse of elements from base types in derived types.

```
type cold[state public hi bye; randevue : hi bye;];
type hot[state public hi kiss bye;
         randevue : hi kiss bye;];
type meet : cold, hot[]; //type meet has all the elements
                         //from type hot and cold
```

initial symbol

Symbol sent to objects before object lifecycle begins. Defined with the begin directive.

```
begin awake;
```

internal state

An internal state from the parsing tables generated by the MUSKOX compiler from the type grammar for use by the state transition engine. Internal states shall not be confused with the object states. Internal states are described in the .v file generated by the MUSKOX compiler (with -v flag). Internal states also appear in the state transition log (mskx.log) recorded by the engine at runtime.

internal wait state

An internal state where the engine will wait for the next symbol to arrive in order to decide which one of two or more actions to undertake before reducing the parsing stack.

keyword

An identifier with a defined name and meaning. It is not used for identifiers. In the MUSKOX documentation keywords are printed in **boldface**.

```
begin awake;          // begin is a keyword
```

LIFE

Builtin reachable symbol. Symbols on the lefthand side of rules with reachable targets are reachable.

```
type dorm
[
    state sleep;          // unreachable symbol
    LIFE : awake;         // reachable symbol
];
```

lifecycle

Object lifecycle.

lifecycle management phase

A loop at the end of the `main()` function of the ARM that dispatches symbols from the loqal queue by invoking the state transition engine on behalf of different objects.

lifecycle member

A member from the `typeProto` class that controls the lifecycle of an object.

link

Named and typed handle defined within a type to link child objects.

```
type child[];
type parent(child l)[];     // child will be linked thru
                            // the link l of type child
child c;                    // this is a child object
parent(c) p;                // this is a parent object
```

macros

Substitution syntax that renames public symbols from base types in derived types.

```
type base[state public oldName;];
type derived : public base <oldName=>newName>[];
```

marked symbol

Symbol of the form `n@l` that denotes state or message `n` from a child linked by link `l`, or symbol of the form `n@?` that denotes state or message `n` from a child linked by any link. Used for automatic state change notification.

```
LIFE : starIsBorn@c;
```

mapped spec

The main file generated by the MUSKOX compiler. Contains C++ classes, objects, and numeric parsing tables generated by the MUSKOX compiler from MUSKOX types and objects.

message

Symbol sent to objects, but that cannot be a target of a rule. Defined with the **message** keyword.

```
message Hello;
```

MUSKOX compiler

The MUSKOX compiler maps design specifications written in MUSKOX metalanguage into efficient C++ programs. An application is designed from objects whose lifecycles are defined in terms of state transitions and inter-object communication, and converted into the Application Runtime Monitor with the MUSKOX and C++ compilers.

MUSKOX log file

File with the extension .log where the engine records the lifecycles of all the objects from the ARM.

MUSKOX metalanguage

A metalanguge of the MUSKOX Software Engineering Tool.

MUSKOX process

A collection of MUSKOX objects executing their lifecycles.

MUSKOX Software Engineering Tool

MUSKOX tool includes MUSKOX metalanguage, MUSKOX compiler, and runtime system.

non-blocking queueing mechanism

Objects sending symbols to others object continue their lifecycles without waiting for the symbol to be delivered.

notification

Mechanism in the State Transition Engine and framework that automatically notifies parent objects about symbols shifted on the stack by a child. The child does not need to invoke a communication instruction or communication method for the notification.

```
type child[ state public starIsBorn; LIFE : starIsBorn;];
type parent(child c)
[
      LIFE : starIsBorn@c
                { cerr << "\nThe star is born!";};
];
```

numeric arrays with parsing tables

Generated by the MUSKOX compiler from type grammars of non-abstract types. They encode LR1 automaton, or the state machine, that control the work of the State Transition Engine at runtime.

object

Instance of a MUSKOX type. See static object and dynamic object.

object definition

Defines static objects as instances of MUSKOX types.

```
type soundCard[];     // MUSKOX type soundCard
soundCard s1;         // object s1 -- instance of a type
```

object declaration

Declares names of objects without creating an instance, to define objects that recursively refer to each other as parents and children.

```
type a(a 1);
declare a o1;         // declaration of object o1
a(o1) o2;             // definition of object o2
a(o2) o1;             // definition of object o1
```

object lifecycle

Order of state changes, received messages and execution of actions, triggers, and reductors of MUSKOX objects.

object lifetime

Time an object is accessible. Static objects are accessible during the entire runtime, while dynamic objects are created at runtime and accessible until deleted.

object state

Declared name that constitutes a step in object lifecycles. States are defined for types and may comprise state transition rules.

```
type binarySwitch
[
     state on off;
     LIFE : <on | off>+;
];
```

parent

Object connected to one or more child objects through links.

```
type child[];
type parent(child link)[];
child c;                      // this is a child object
parent(c) p;                  // this is a parent object
```

parsing error

Runtime occurrence when a symbol arrives that cannot be shifted on the parsing stack, either because it is defined but not expected or not defined within a type.

parsing stack

Stack for MUSKOX objects where the State Transition Engine keeps the symbols before reducing the stack by matching the top of the stack with the righthand side of a rule.

process

See MUSKOX process.

qualifiers

Qualifiers assign properties to defined symbols, states and messages. Qualifiers are:

```
used     public     private    hidden    skip    left
right    nonassoc   copy       delay
```

queue

See symbol queue.

queueing mechanism

See non-blocking queueing mechanism.

reachable symbol

Symbol in the righthand side of a rule with a reachable target. Only reachable symbols appear on the parsing stack at runtime. Built-in state **LIFE** is a reachable symbol by definition.

reduce/reduce conflict

A kind of ambiguity in the state transition rules where two or more reductions are possible. Reduce/reduce conflicts are diagnosed by the MUSKOX compiler as errors.

reduction

Step in State Transition Engine execution when it reduces the accumulated stack of states and messages by matching the top of the stack with the righthand side of a rule.

reductor

C++ code that the State Transition Engine executes before a reduction of any rule. Defined with a **reductor** directive.

```
reductor { cerr << "This is a reductor!";};
```

regular expression

Shorthand syntax for repetitive sequences of symbols, actions, and communication instructions that appear in the righthand side of the rules.

```
LIFE : <on | off>*;  // any sequence of states on or off
```

regular operator

Defines a number of times a regular expression may be repeated.

```
+       one or more times
*       zero or more times
-       one or zero times
none    exactly one time
```

repetition rule

A rule with one or more targets that remain unchanged after state transitions defined in the righthand side.

```
repeat <*> : Shake;    // any state remains unchanged
                       // after receiving a message Shake
```

resolution phase

Execution of the initial portion of the `main()` function of the ARM during which all major pointers between the components of the ARM are resolved.

righthand side of the rule

Sequence of states, messages, actions, and communication instructions that objects undergo before reduction of a rule and transition to the target of the rule.

```
LIFE :                    // target of the rule
     open doJob close;    // right-hand side of the rule
```

rule

See state transition rule.

selection rule

A rule with one or more targets that is quivalent to a number of rules where the targets are defined as last symbols in the righthand sides:

```
select < open closed > : Open openAction;
```

equivalent to

```
open   : β open;
closed : β closed;
β : Open openAction;
```

shift

Step in State Transition Engine execution when it shifts a symbol on the parsing stack.

shift/reduce conflict

A kind of ambiguity in the state transition rules where both shift and areduction are possible.

Software Engineering Tool

A tool to engineer software applications. MUSKOX, GUIs, class libraries, and C++ compilers are examples of Software Engineering Tools.

spec

A logical sequence of MUSKOX elements prepared in a source file.

stack

See parsing stack.

Standard Framework

A framework supplied with the MUSKOX Software Engineering Tool. Developers may use or enhance it for their application, or replace it with their own.

state

A MUSKOX element defined within a type. Objects which are instances of the type transition through states during their lifecycle.

state change

See state transition.

state machine

A state machine is comprised from internal states that are generated by the MUSKOX compiler from a given type and stored in the numeric arrays in the generated .C file. State machine controls operations of the state transition engine at runtime.

state transition

Shift of a state on the parsing stack by the State Transition Engine. State transitions occur when a state is received from a symbol queue. State transition to a target state of a rule occurs after the rule reduces the stack.

State Transition Engine

Part of the ARM responsible for executing object lifecycles. It accumulates states and messages on the parsing stacks of objects, and reduces the stacks by matching its top with a rule.

state transition log

Log of state transitions and communicated symbols. Maintained by the framework and State Transition Engine.

state transition rule

Defines the order of received symbols, state tarnsitions, actions, and communication instructions that constitute object lifecycles.

```
LIFE : open doJob close;
```

static object

Object defined in a MUSKOX spec. Static objects persist during the runtime of an application.

```
disk d1;          // static object d1 of type disk
```

syntax diagram

A graphical picture that explains the syntax of a MUSKOX element in the MUSKOX Reference Guide.

symbol

State or message defined within a type.

```
state stable;
message Hi Bye;
```

symbol queue

Queue of symbols, states and messages delivered to objects by the framework from the same or other objects, or from outside of the system.

symbolic communication

Communication by messages and states. Communication by states is useful in debugging and rapid prototyping.

synchronization

Method for coordinating lifecycles of multiple objects, by either sending symbols or by notification.

system-generated identifier

A symbol name in a derived type that the MUSKOX compiler generates for a private symbol from a base type.

system-generated rule

A state transition rule that is generated by the MUSKOX compiler for a regular expression, repetition, and selection rules.

system-generated state — A state generated by the MUSKOX compiler for notification, regular expressions, repetition and selection rules.

target — State specified on the lefthand side of a rule.

```
LIFE : Hi kiss Bye;     // LIFE is the target of the rule
```

thread — See object lifecycle.

trigger — C++ code or communication instruction executed by the State Transition Engine after a symbol is shifted on the parsing stack. Defined with the trigger directive.

```
trigger { cerr << "This is a trigger";};
```

type — MUSKOX type is a C++ class augmented by the lifecycle containing states, messages, rules, triggers, reductors, etc.

```
type file                    // C++ portion
{
   public:
   char *Name;
}
[                            // MUSKOX lifecycle portion
   LIFE : <open | closed>*;
];
```

type declaration — List of types that need not be defined prior to declaration, to define types with recursive links.

```
type parent;                // type declaration
type child(parent p)[];     // type definition
type parent(child c)[];     // type definition
```

type definition — MUSKOX type to define MUSKOX objects -- instances of types.

```
type muskox[];          // a type with no lifecycle
```

type exception — Type or list of types in link definitions specifying type objects that may not be linked by this link.

```
type parent(child ~badBoy c)[]; // no children
                                // of type badBoy
```

type grammar — Set of all the symbols, rules, triggers, reductors, virtual grammars, and other MUSKOX elements that describe object lifecycles.

type hierarchy — Hierarchy of MUSKOX types and C++ classes built with inheritance much like class hierarchy in C++.

typeProto — Virtual base class of a MUSKOX type that contains data and function members that support object lifecycles.

USSA — Precursor of the MUSKOX Software Engineering Tool.

virtual grammar

Named set of rules, triggers, and reductors from a base type that may be redefined in a derived type.

```
type meeting
[
   state public beginning;
   message public Hi;
   virtual greeting          // virtual grammar
   [
      begining : Hi;
   ];
];
```

virtual grammar inheritance

Elements from virtual grammars defined in base types may be redefined by elements of a virtual grammar with the same name from the derived type.

```
type randevue : public meeting
[
   state public beginning kiss;
   message public Hi;
   virtual greeting         // virtual grammar
   [
      beginning : Hi kiss;  // redefine the rule
   ];                       // from type meeting
];
```

virtual reductor

A shorthand notation for a virtual grammar that contains a single reductor:

```
virtual vred reductor { cerr << "\nHello World!";};
```

virtual rule

See virtual state transition rule.

virtual state transition rule

A shorthand notation for a virtual grammar that contains a single state transition rule:

```
virtual vrule LIFE : Hello+;
```

virtual trigger

A shorthand notation for a virtual grammar that contains a single trigger:

```
virtual vtrig trigger { cerr << "\nHello World!";};
```

virtual type

A type defined with a **virtual** keyword. State transition rules from a virtual type that do not belong to an explicitly defined virtual grammar belong to a virtual grammar implicitly defined by the MUSKOX compiler with that type's name.

```
virtual type a[ LIFE : Hello+; ];
```

7. MUSKOX BACKGROUND

This section is provided for those readers that want a deeper understanding of the MUSKOX background in the formal languages and in event-driven systems.

7.1 MUSKOX AND FORMAL LANGUAGES

At runtime an object accepts a flow of symbols that constitutes a sentence from a language defined by the object's type grammar. The symbols represent external and internal events. The State Transition Engine essentially parses this sentence and executes actions during parsing. The grammar's rules control the application logic, while the actions execute the code of the application.

COMMON FEATURES:

- The MUSKOX metalanguage is a dialect of the Backus-Naur Form for defining context-free grammars
- A type grammar specifies a formal language where terminals correspond to messages, and nonterminals correspond to states
- The State Transition Engine is an implementation of a LR1 parser
- The MUSKOX compiler is a LR1 parser generator that converts context-free type grammars into the numeric arrays with LR1 parsing tables

DIFFERENCES:

There are several major differences between MUSKOX parser and 'traditional' parsers:

- In MUSKOX objects generate the flow of states and messages when parsing the same flow, while traditional parsers accept symbols without affecting the flow
- The MUSKOX compiler processes any number of context free grammars, and generates multiple parsers at compile time
- The runtime framework may execute any number of parsers simultaneously
- The MUSKOX State Transition Engine may accept both terminals (messages) and nonterminals (states)
- MUSKOX type grammars are associated with C++ classes and allow inheritance and redefinition of rules

7.2 MUSKOX AND EVENT-DRIVEN SYSTEMS

SIMILARITIES:

MUSKOX state transition rules specify processing of external and internal events in common with event-driven systems:

- External events outside of the MUSKOX framework are delivered at runtime as symbols, both states and messages
- In response to events the MUSKOX Framework executes C++ programs and undergoes state transitions according to state transition rules
- The framework may generate internal events, states and messages with which objects communicate between themselves, and synchronize event processing

DIFFERENCES:

There are several major differences between MUSKOX and other event-driven systems:

- The MUSKOX state transition engine uses a stack of events (the parsing stack) for such applications as transaction processing, while traditional event-driven systems are usually based on a finite state machine that keeps no history of state transitions
- MUSKOX metalanguage still allows finite state machines if only the following kind of state transition rules is used in MUSKOX types:

```
newstate : currentState1 messageArrived1
         | currentstate2 messageArrived2 ...
```

- Both MUSKOX messages and states may be sent to objects, allowing for efficient communication, debugging, and rapid prototyping, while traditional event-driven systems strongly differentiate between an internal state and an external event (message)
- MUSKOX inheritance model permits adding new events (states and messages), state transition rules, and redefining existing rules, while event-driven systems usually allow only state machines with hierarchical states
- The MUSKOX metalanguage is a lightweight addition to C++ and may easily be extended to other programming languages, while other event-driven systems are implemented either as a fullfledged programming language, a library of subprograms or a set of programming conventions
- MUSKOX compiler converts states and messages into nonnegative numbers, while some other event-driven systems represent states as types or templates, and messages as subprograms