An Overview of the Next Scripting Toolkit

Gustaf Neumann and Stefan Sobernig

{firstname.lastname}@wu.ac.at

Tcl/Tk 2011 Conference, October 2011

Abstract

This paper introduces the Next Scripting Framework (NSF) and the Next Scripting Language (NX). The paper presents features such as the definition of object systems, parametric objects, initialization and interfacing to object states, creating object behavior, and designing object interfaces and interactions. Along the way, we review some syntactic additions and developer support tools for developing NSF/NX programs. Our goal is to provide a comprehensive overview of the NSF/NX features, including hands-on code examples, by comparing NX to its next relative XOTcl.

A Toolkit for Developing A Family of OO Languages

The Next Scripting Framework (NSF) and the object system NX have been developed between 2008 and 2011 at the Institute for Information Systems and New Media of the Vienna University of Business and Economics. These systems continue a research line and a development effort, starting in the late 90s, to develop better language support for adopting OO Design Patterns, for managing program variability by first-class abstractions (e.g., aspect and feature modularization), and for creating different object-oriented languages in Tcl, as well as special-purpose application languages; e.g., embedded, textual DSLs [15]. As the first code artifact, XOTcl was presented in 2000 [4] and introduced novel language constructs: filters, as well as per-object, per-class, and transitive mixin classes [7]. XOTcl heavily influenced the design of TclOO [5], which is in many respects a simplified and streamlined descendant of XOTcl.

The Next Scripting Framework (NSF) generalizes many ideas of XOTcl. NSF allows for fully scripted definitions of object systems, while preserving (and even improving) the performance properties of C-based implementations. For example, the scripted NSF implementation of XOTcl 2.0 is significantly faster than the C-based XOTcl 1.6 implementation [3].

NSF lets the Tcl programmer create several object systems in a single interpreter. Object systems are initially created without any predefined behavior (methods), granting the object system designer (Tcl developer) the full freedom of defining and naming method interfaces. With scriptable object systems and new composition techniques (e.g., method aliasing), NSF adds to Tcl's support for language-oriented programming [8].

While XOTcl 2.0 is designed for backward compatibility with XOTcl 1.* scripts, the Next Scripting Language (NX) is the result of an extensive re-design and perfective refactoring of XOTcl. This further development builds on the experience of several large-scale development projects (i.e., several hundred thousand lines of Tcl/XOTcl code, 10+ developers, etc.). The NX language is designed to ease language learning by novices (e.g., by using "mainstream" terminology, higher orthogonality of method interfaces, smaller core interfaces), to improve maintainability (e.g., preventing common errors) and to encourage developers to create better structured programs. Providing different types of interface abstraction, code evolution and collaborative development between several developers are facilitated.

The remainder of this paper expands on key features of the NSF/NX programming toolkit. In this feature presentation, we want to stress the advancements achieved since our Tcl'09 paper [3]. First, we introduce some basics of the object system model (in particular, entity and relationship types) underlying any NSF-based language. In a subsequent step, we guide through the major contributions: Concrete syntax enhancements (scripted init-blocks, prefixes for instance variables and methods), new language abstractions (method ensembles, method aliases, properties), and added language expressiveness (object parametrization, parameter types). We also sketch out the developer support provided by the NSF/NX toolkit, including DTrace integration and a memory debugging facility, generator support for developing Tcl/C APIs and extension libraries, a functional testing environment (nx::test), and a documentation generator (nxdoc). We conclude by reporting performance data collected for the NSF/NX language runtime.

Scripted Definition of Object Systems

NSF offers a low-level API providing a small set of primitive commands to define the behavior of tailored object systems. An object system is formed from a subset of the (extensible) base features with free naming support. The notion of object systems stresses objects as the first-class entities. Objects can be related differently, including meta-class/class, class/instance, mixin, and composition relations [4]. For managing object states, APIs of different expressiveness and complexity (primitive setter/getter commands, accessor/mutator methods, slots) can be adopted. For defining object behavior, methods can be defined for various scopes (e.g., object, class, mixin) and ad-

vanced forms of implementation reuse (e.g., method aliasing of Tcl procs and Tcl/C commands) are available in addition to forwarders. Method properties such as redefinition and call protection can be specified. Parametric objects and methods can be realized using a unified parametrization infrastructure, equipped with non-positional and positional parameters and parameter type annotations.

Once defined, multiple object systems can coexist in a single Tcl interpreter, the object systems can be used interleaved in a script. For example, NX is provided as a purely scripted Tcl package (loadable via "package require") in the same way as the backward-compatible XOTcl 2.0 object system.

The Next Scripting Framework (NSF) provides a set of about 30 language-programming primitives in the `nsf` namespace. The primitive `nsf::objectsystem::create` allows for declaring a pair of root objects for an NSF object system: a root class (first argument) and a root meta-class (second argument).

Listing 1: A minimal NSF Object System

```
# Create an object system with the base classes named "myObject" and
# "myClass"
nsf::objectsystem::create myObject myClass

# Bind a pre-existing method for creating objects from the methods
# pool in "nsf::methods" as "+" to "myClass". After this method is
# registered, every class/meta-class of this object system can use "+"
# to create objects or classes.
nsf::method::alias myClass + nsf::methods::class::create

# Bind a pre-existing method for deleting objects from the methods
# pool in "nsf::methods" as "-" to "myObject". Once this method is
# defined, every object of this object system can be deleted using
# "-".
nsf::method::alias myObject - nsf::methods::object::destroy

# Create an application class using the method "+":
myClass + C

# Create an instance of the application class:
C + c1

# Delete the instance using the method "-":
c1 -
```

As can be seen from Listing 1, the names of the base classes (`myObject` and `myClass`) are provided to the command `nsf::objectsystem::create` as the first two arguments. The root objects determine elementary relationship types between the objects living in a given object system and describe common behavior for all objects. The creation command for the object system covers several tasks: To begin with, the memory stores for the root objects are created. Once allocated and reg-

istered as Tcl commands, the root objects are put into elementary relations to each other (e.g., instance-of and superclass/subclass relations; see below). Finally, the essentially behavior-less root objects (i.e., their empty method records) can be populated with behavior by the language designer.

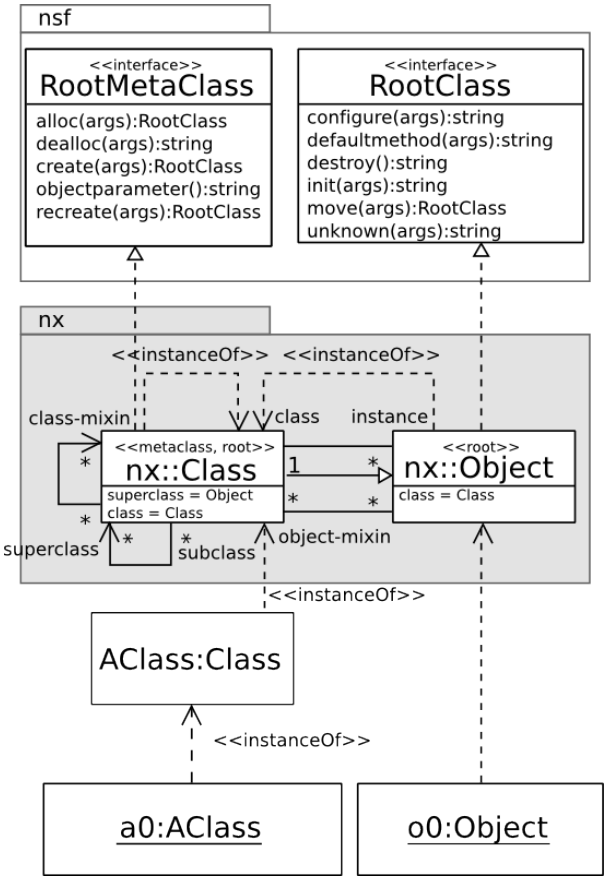

Figure 2. The NX Object System

While the bare root objects do not carry any predefined or built-in methods accessible at the script level, the NSF engine requires the root objects to support basic lifecycle operations (e.g., object creation, deletion, recreation etc.). These methods might not only be called in the script, but also from within the NSF engine. For these cases, one can optionally bind methods to the system callbacks during the definition of the object system (not shown above). In this sense, the root objects implement interfaces required by the NSF engine (see `RootMetaClass` and `RootClass` in Figure 2).

When implementing these two required interfaces, the language designer has considerable degrees of freedom: First, one can choose custom names (selectors) for the system methods. Second, one can bind either predefined or custom method implementations to these selectors. Third, upon declaring the object system by

`nsf::objectsystem::create`, one can define custom default bindings for the system methods without exposing them as accessible methods.

```
nsf::objectsystem::create myObject myClass
nsf::method::alias myClass  + ::nsf::methods::class::create
nsf::method::alias myObject - ::nsf::methods::object::destroy

nsf::is class myObject      ; # --> 1
nsf::is metaclass myObject  ; # --> 0

nsf::is class myClass       ; # --> 1
nsf::is metaclass myClass   ; # --> 1

nsf::relation myObject class       ;# --> ::myClass
nsf::relation myObject superclass  ;# -->

nsf::relation myClass class        ;# --> ::myClass
nsf::relation myClass superclass   ;# --> ::myObject
```

The creation of an object system establishes characteristic and mutual ties between the root meta-class `myClass` and the root class `myObject`. Most importantly, `myObject` is defined as an instance of `myClass` (the class of `myObject` is `myClass`), and `myClass` is a subclass of `myObject` (the superclass of `myClass` is `myObject`). Therefore, every class is an object and inherits the general object behavior.

This relational triad between root meta-class and root class underlies any NSF object system and is automatically established by `nsf::objectsystem::create`. A language designer can obtain the same relational setting by declaring the relations explicitly, using the NSF primitive `nsf::relation`.

Listing 3: System Methods Specification for the NX Object System

```
namespace eval ::nx {

  nsf::objectsystem::create ::nx::Object ::nx::Class {
    -class.alloc {alloc ::nsf::methods::class::alloc}
    -class.create create
    -class.dealloc {dealloc ::nsf::methods::class::dealloc}
    -class.objectparameter objectparameter
    -class.recreate {recreate ::nsf::methods::class::recreate}
    -object.configure configure
    -object.defaultmethod {defaultmethod ::nsf::methods::object::defaultmethod}
    -object.destroy destroy
    -object.init {init ::nsf::methods::object::init}
    -object.move move
    -object.unknown unknown
  }

}
```

NSF defines a set of about 30 primitive commands in the `::nsf` namespace for further defining the object system. For application developers, however, the necessary functionality offered by the NSF primitive commands is exposed by the object system (e.g., the `info` method for introspection) directly.

The NX object system (see Figure 2) is entirely defined using these language-programming primitives. Listing 3 depicts the relevant script fragment for creating the NX root objects (`nx::Object`, `nx::Class`), as well as for tailoring the provided system method interfaces.

The NX Concrete Syntax

Scripted Init-Blocks - Defining Objects Block-wise

In the tradition of nesting evaluable Tcl scripts as definition units (e.g., proc bodies, looping constructs, namespace scripts), NX objects can evaluate scripts in their context upon request or upon initialization. The scripted init-blocks are evaluated at the end of object initialization and are typically used for defining variables, properties and methods. A block-wise notation helps avoid redundancy (i.e., tediously repeated object names) and allows for grouping related declaration statements.

Consider a bare example. Instead of defining a class and its structural features (i.e., relations, properties, and methods) via separate Tcl commands ...

```
nx::Class create ASuperClass
nx::Class create AClass

AClass superclass ASuperclass
AClass property aProperty
AClass public method aMethod {} {...}
```

one can specify a script for every object/class definition which is evaluated in the context of the newly created entity:

```
nx::Class create ASuperClass

nx::Class create AClass -superclass ASuperClass {
   :property {aProperty 0}
   :public method aMethod {} {...}
}
```

The `create` method accepts the name of the entity to be created (here `AClass`) and optional, non-positional parameters for configuring the entity; referred to as *object parameters*. After the object parameters, an optional script might be provided which is called the *init script*. In this example, all commands in the init script are prefixed by a single colon, which means that they denote methods dispatched on the current object (here `AClass`). This is achieved by using a special-purpose command resolver [3].

Scripted init-blocks are equally available for declaring all kind of objects, i.e., direct instances of `nx::Object` or instances of arbitrary application classes.To create instances of the previously defined class `AClass`, one can write:

```
AClass create a0
AClass create a1 -aProperty 10
AClass create a2 {
   :public method foo {} {...}
}
```

While the instance `a0` is created without object parameters (using just the defaults), the instance `a1` is initialized by object parameters, and `a2` uses a scripted init block for defining an object-specific method `foo`.

The Colon Prefix - Shortcutting Self Calls and Self-Variable Access

By leveraging Tcl's variable and command resolver infrastructure, NSF introduces colon-prefixed names for referencing instance variables and for specifying method calls with implicit receivers for little syntactic overhead. The colon prefix refers to the current object for the scope of scripts evaluated in an object's context (e.g., in init scripts or in method bodies).

```
AClass create a2 {

   set :x 1; # set an instance variable named "x"

   :public method foo {} {
       set x 1    ; # set a method-scoped variable
       set :x 1   ; # set an instance variable
       set ::x 1  ; # set a global variable
       incr :x    ; # access an instance variable
       puts "var x value ${:x}"; # refer to value of an instance variable
   }

   :foo
}
```

In the above listing, each colon-prefixed variable reference resolves to an instance variable named `x` stored with the object `a2`. When requesting instance variable substitution, the dollar sign must be preceded by the colon-prefixed variable name protected by a pair of curly braces, for example: `${:x}`. A colon-prefixed command name (such as `:public`) corresponds to an invocation of a method of the same object. For example, `:foo` corresponds to `my foo` in XOTcl.

Slim Method Set - Easing API Learning

Each NSF object system provides a core API through its base classes. The perceived usability [2] of APIs is affected by various cognitive properties, including the API's conceptual chunks needed for frequent programming tasks (e.g., introspection) and the penetrability of an API. An ultimate design goal was therefore to keep the core interface of NX as concise and as consistent as possible. As a result of this design effort and new implementation techniques being available (e.g., extensible method ensembles), the NX core API consists of only 44 methods, as compared to 124 in XOTcl, while exhibiting a functional superset of the XOTcl core API.

Table 1. Comparison of the Number of Predefined Methods in NX and XOTcl

	NX	XOTcl
Methods for Objects	20	51
Methods for Classes	3	24
Info-methods for Objects	15	25
Info-methods for Classes	6	24
Total	**44**	**124**

In addition to the reduced interface sizes, the NX core APIs also benefit from the capacity of creating method interfaces in a hierarchical manner. The figure below sketches the tree-like structure of the `info` introspection available for all instances of `nx::Object`. Each sub-level of the hierarchical interface (e.g., `callable`, `has`, `filter`, and `mixin`) groups introspection operations which relate to the same language construct to be introspected (e.g., mixins or filters) or which identify a particular introspection scope. For example, `info callable` refers to the methods dispatchable on a given object rather than the ones defined by it (`info methods`). Hierarchical method interfaces allow the language or application developer to define working frameworks [2] within an API. At the same time, the hierarchically organized interfaces can still be extended and refined by standard means of method combina-

tion (e.g., mixin classes) at each sub-level. This API structuring technique is the result of using method ensembles(find details below).

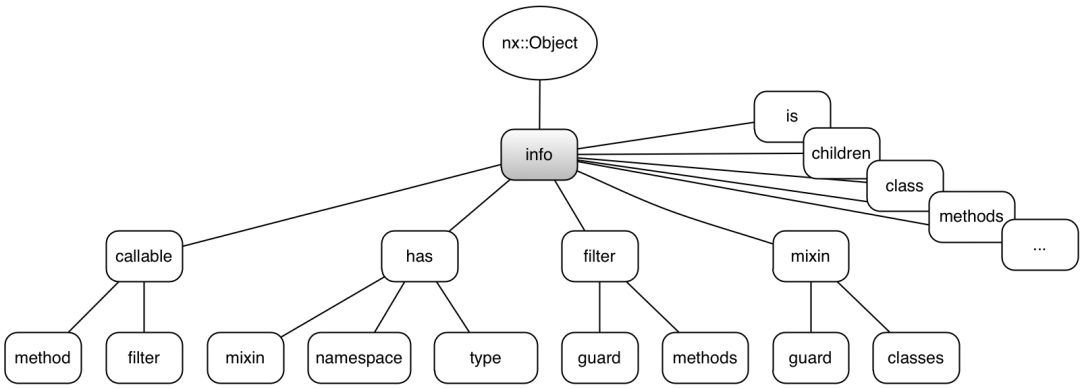

Figure 4. Hierarchical Method Interfaces: An Excerpt from the 'info' Method Ensemble

Parameter Types and Parameter Options - Constraining and Transforming Parameter Values

Tcl provides the command `string is` to check whether a provided string has certain properties, i.e., whether it can be converted into an internal representation with a certain value format. NSF extends this value checking for specifying method parameters and method return values, as well as object interfaces. A method is specified by a method signature, i.e., the number of method parameters (in/out), their names, and value constraints defined over the permissive arguments. An object is configured by object parameters.

Value constraints for method and object parameters can be specified with built-in and custom defined parameter type checkers. They apply to both positional and non-positional parameters. The range of built-in constraints includes object-type checks and predefined Tcl value classes. Table 2 below presents selected examples of parameter types and options. Additionally, custom defined value checkers can be provided by defining special-purpose methods.

For all types of value checkers, parameter options can be specified to define the multiplicity class and the optionality of the parameters. Moreover, parameters can be turned into method and forwarder dispatches, using disposition parameters. For multivalued object parameters, an incremental getter/setter API is available, offering the per-element operations `add` and `delete`.

These provided value checkers can also be used to perform representational transformations on parameter values (e.g., normalizing values). This syntactic value checking

can be en- or disabled for the scope of an interpreter; in the sense of an optional representational "type system" [1].

Listing 5: Parameter Types for Arguments and Return Values

```
nx::Class create C {

  # Define method "set" with an optional positional parameter "value":
  :public method set {varName value:optional} {
    # ....
  }

  # Define method "foo" with a non-positional parameter "opt" having a
  # default value and a positional parameter "x" with the value
  # constraint "integer":
  :public method foo {{-opt true} x:integer} {
    # ....
  }

  # Define a method "bar" with a non-positional
  # parameter "objs" carrying the value constraint "object" under the
  # multiplicity class "1..n" and a positional parameter "c" with value
  # constraint "class" for a multiplicity of "0..1":
  :public method bar {-objs:object,1..n c:class,0..1} {
    # ...
  }

  # Bind the Tcl command ::incr as a method (an alias) to the class and specify
  # that it always returns an integer value:
  :public alias incr -returns integer -frame object ::incr

  # Define a forwarder that has to return an integer value:
  :public forward plusOne -returns integer ::expr 1 +
}
```

Value checking is fully integrated with the argument parser and the error handler for scripted and for C-implemented methods. For C-implemented methods, value checking provides the internal representations (e.g. integers, boolean, objects, classes, etc.) as arguments to the underlying C functions [3]. This greatly helps implement C extensions, such as the MongoDB binding described later in this paper.

Table 2. Thumbnail Descriptions of Common Parameter Types and Parameter Options

Parameter type/option	Description
	Value constraints
integer	The argument must be a 32-bit Tcl integer (`string is integer`).
boolean	The argument must be one of the acceptable Tcl boolean values, e.g. `0`, `1`, `true`, `false` (`string is boolean`).

Parameter type/option	Description
object ? type=className ?	The argument must refer to an existing object (i.e., an instance of the root class `nx::Object`). If the `type` option is provided, the object's class must correspond to an existing class *className*.
class ? type=metaClassName ?	The argument must refer to an existing class (i.e., an instance of the root meta-class `nx::Class`). If the `type` option is provided, the class' meta-class must correspond to an existing meta-class named *metaClassName*.
	Multiplicities
0..*, 0..n	Specifies that the argument can be either an empty list (i.e., `""` or `[list]`) or a list with any number of elements (unbound cardinality). If the argument is a non-empty list (element cardinality > 0), each element is then tested against the value constraint specified.
0..1	Specifies that the argument can either be an empty list (i.e., `""` or `[list]`) or a list with exactly one element (cardinality: 1). If the argument is a non-empty list (element cardinality > 0), the element is then tested against the value constraint specified.
1..*, 1..n	Specifies that the argument must be a non-empty list with an unbounded number of elements (cardinality > 1). Each element is then tested against the value constraints specified.
	Requiredness/Optionality
required	An argument for the parameter must be provided. Note: Positional parameters are considered required implicitly.
optional	An argument for the parameter may be omitted in the arguments vector. Note: Non-positional (named) parameters are considered optional implicitly.
	Disposition
alias ? method=methodName ?	The parameter specifies a method dispatch to a method identified by the parameter name or, if the `method` option is provided , to a method *methodName*. An unqualified name resolves to a method for the scope of the called object.
forward method=forwardSpec	The parameter specifies a forward dispatch, according to the mandatory `method` type which contains the forward specification *forwardSpec*.
	Various
switch	The parameter is specified as a flag, i.e., a non-positional parameter which does not accept an explicit argument. If the flag is provided, the default value (`0` for `false`) is toggled. The default value can be set explicitly to change the toggle direction.
incremental	The object parameter representing a multivalued instance variable should be mutable through per-element ("incremental") setter methods, including methods for adding and deleting single elements.

Object Parameters - Configuration Interfaces for Objects

Like method signatures declaring positional and non-positional parameters with default values and value constraints, NX provides parameters for initializing and configuring objects and classes. The parametric object interfaces are derived from the class definitions. In conventional OO languages, object creation and initialization are realized by chained constructor methods, risking unwanted interactions in classification hierarchies (e.g., common constructor anomalies [9]). The less ambiguous object initialization through object parameters and scripted init-blocks complements the use of constructors.

Recall the classic example of a compositional anomaly resulting from pairing constructor chaining and dynamic method binding in a class hierarchy. The following code listing reproduces an example for creating partially initialized objects for XOTcl, adopted from [12].

```
xotcl::Class create A
A instproc init args {
  # 2) Invoke method "m", dispatching to B.m()!
  my m
}
A instproc m {} {
  # ...
}

# A subclass, possibly defined by a different module (e.g., Tcl package)
xotcl::Class create B -superclass A -parameter {b}
B instproc init {s} {
  # 1) Pass control to the superclass constructor
  next    ; # dispatching A.init()
  # 3) Initialize and define the instance variable "b"
  my instvar b
  set b $s
}
B instproc m {} {
  # 4) Returning instance variable 'b', which is expected to be
  # already initialized and defined
  my instvar b
  return $b
}

B create b1 "ZAP!"; # --> can't read "b": no such variable while executing "return $b"
```

The numbering of the comments (1, 2, 3, and 4) reflects the "intended" unfolding of the control flow during the creation of an instance of B. The anomalous behavior manifests in terms of step 3 effectively occurring after step 4. This is due to the dispatch to *m*, which is contracted by the superclass constructor `A init`, causing an preemptive attempt to access of `B`'s instance variable `b`, yet to be initialized and defined in the subclass constructor `B init`.

This is only one example of various kinds of constructor anomalies discussed in [9]. A further critical kind of anomalies is that construction protocols, though automatically inherited down a class hierarchy (at least in NX and XOTcl), can be easily breached — maybe intentionally, maybe accidentally — by simply omitting a `next` in a subclass constructor. NX, as well as XOTcl, are even more vulnerable to such anomalies due to the considerable degrees of freedom during object configuration (e.g., dispatching to `init` or accessor methods in arbitrary orders) and due to the compositional complexity incurred by mixin classes and transitive mixins.

The object parameter facility in NX relaxes this vulnerability to constructor-based parametrization anomalies considerably. Rewriting the above example in NX yields, for example:

```
nx::Class create A {
  :method init {} {
    :m
  }
  :public method m {} {
    # ...
  }
}

nx::Class create B -superclass A {
  :property b:required
  :public method m {} {
    return ${:b}
  }
}

B create b1 -b "ZAP!"
B create b2; # --> required argument 'b' is missing, should be: ::b2 configure -b ...
```

Object parameters provide means for discriminating between four separated stages when constructing objects:

1. *Creation:* This is a class-side event, with the operations for allocating a memory store etc. being performed in the scope of the instantiating class.
2. *Parametrization:* At this stage, the argument vector passed to the object creation procedure (i.e., `-b "ZAP!"`) is evaluated against the object parameter specification of the newly created instance. This specification represents the concatenation of all object parameters (e.g., `A property b:required`) going up the entire inheritance path of the instance's class. The parameter specifications can also contract the mandatoriness or value ranges of parameter values, along with default values etc.
3. *Setup by Init Script:* After having completed the parametrization stage, the object is fully initialized as stipulated by the object parameter specification. The evaluation of the init script block is performed to allow for continued set-

up of the newly constructed object. This step can only be performed once, i.e., at construction time, as the init script is not preserved.

4. *Setup by Constructor:* Finally, the chain of `init` methods provided is invoked upon. Note that in NX, the `init` methods do not receive any intermediary results of previous object construction or residuals of the initial vector of construction arguments as input arguments. In NX, constructor methods are therefore not equipped for initializing the initial state of an object. Still, they serve as important extension points during object construction.

Object Variables and Properties - Defining Object State

NX supports defining instance variables with and without accessor methods. While internally accessible instance variables are defined via the method `variable`, externally accessible instance variables are equipped with accessors (setter/getter methods). In addition, so accessible instance variables can also be exposed as object parameters by the object interface. Instance variables with accessors are created using the `property` keyword. Value checkers can be specified for instance variables defined via `variable` and via `property`. Properties can also be accessed through an incremental getter/setter interface (`add`, `delete`). The following listing gives three showcase examples, including the specification of default values and parameter types with `property` and `variable`, respectively:

```
nx::Class create AClass {
  :property {aProperty:integer 0}
  :variable aVariable:integer 0
  :property {multiProperty:1..*,integer,incremental 0}
  :create a1
}

#
# property plus setter/getter methods
#
::a1 aProperty; # returns "aProperty" (0) through the so-named getter method
::a1 aProperty 1; # sets "aProperty" through the so-named setter method

#
# variable without setter/getter methods
#
::a1 aVariable; # no getter method: ::a1: unable to dispatch method 'aVariable'
::a1 aVariable 1; # no setter method: ::a1: unable to dispatch method 'aVariable'
::a1 eval {set :aVariable}; # internally, the instance variable is accessible/mutable

#
# property with incremental setter/getter methods
#
::a1 multiProperty; # returns 0
::a1 multiProperty delete 0; # removes an element from the list
::a1 multiProperty add 1; # adds an element to the list and returns 1
::a1 multiProperty add 2 end; # adds another element and returns "1 2"
```

Methods

Like XOTcl, NX offers open class and open object definitions. This means, for example, that it is possible to define a class or an object without methods and to add methods dynamically at runtime. NX supports scripted and C-implemented methods. Scripted methods are defined via a predefined keyword `method`. When `method` is applied on a class, an instance method is defined (i.e., a method applicable to instances of the class); when `method` is applied on an object, an object-specific method is defined. The method definition can be refined by modifiers such as `public` and `protected` to request call protection and by the keyword `class` to refer explicitly to the class object. One can use `class method` to define methods applicable to the class object. Such methods are sometimes referred to as "class" or "static" methods. Similarly, one can use `class variable` or `class property` to define variables and properties for the class object.

Aliases and Forwarders - Method-Level Reuse

In addition to defining scripted methods as outlined above, NX supports reusing preexisting method definitions for a class or for an object by means of method aliases and method forwarders. For aliasing a method, NX provides the method `alias`. Aliasing means registering a method by a distinct name with an object. This method alias can refer to the implementation of a method of another object/class, a Tcl proc, or even a Tcl/C command.

In NX, the idea of assembling the base class interfaces from a set of core C-implemented commands [3] is extended to a general-purpose aliasing mechanism in NX (not to be confused with Tcl's interp aliases). Method aliases are one foundation of traits and method ensembles (we go into more details in later sections). Aliases serve for bootstrapping an object system and are an essential instrument for object system developers (as presented earlier in Listing 1).

Listing 6: Method Aliases and Method Forwarders

```
nx::Class create C
  :property {a 0}
  :public alias incr -returns integer -frame object ::incr
  :public forward plusOne -returns integer ::expr 1 +
}

C create c1     ;# create instance c1
c1 incr a       ;# increments instance variable "a" to 1
c1 incr a       ;# increments instance variable "a" to 2

puts [c1 a]     ;# outputs 2
```

```
puts [c1 plusOne [c1 a] * 100] ;# outputs 201
puts [c1 a]    ;# outputs 2
```

The `alias` statement in Listing 6 are taken from Listing 5. It defines a public instance method named `incr` of the class `C`, which reuses the implementation of the C-implemented Tcl command `::incr`. The parametrization by `-frame object` has the effect that variable names provided as arguments to the newly defined method `incr` refer to instance variables. Note that all arguments provided to a method alias are always passed unmodified to the underlying command implementation.

A *method forward* is somewhat similar to a method alias except that one can extend and rewrite the provided argument vector. The definition of the method `plusOne` reuses the Tcl command `::expr` and adds `1 +` at the front of the provided argument vector to complete the Tcl expression.

In general, a method forward is more flexible than a method alias, but less efficient. Apart from efficiency, method aliases have another important property: For a method alias, introspection returns the method parameter specification of the alias target (if available). Parameter introspection is not possible for a method forward.

Method Ensembles - Implementing Hierarchical Method Interfaces

The capacity of objects to act as message receivers [3] has been further refined into the concept of ensemble objects and method ensembles. Resembling Tcl's idiom of sub-commands and namespace ensembles, ensemble methods establish hierarchical and compound method names in an extensible fashion. From the perspective of a method client, not only a single but multiple Tcl words are the selectors of a method implementation. As for the method provider, a complex protocol (e.g., introspection through info) can be organized into several related ensemble method implementations.

Central to the compositional feature of ensemble methods is the idea of breaking up otherwise monolithic methods with heavy conditional branching (e.g, extensive switch threading) into distinct units, i.e., ensemble methods [11]. At the same time, the ensemble methods remain grouped by a parent method selector. Consider the following example:

Listing 7: Definition of Ensemble Methods without Language Support

```
Object create o {

  # Define method "foo", the parent method selector:
```

```
    :public method foo {sub args} {
      #
      # Define sub-methods behavior via "switch" statement
      #
      switch -exact -- $sub {
        sub1 {
          # ensemble method 'foo sub1': provide a custom parser for "args"
        }
        sub2 {
          # ensemble method 'foo sub2': provide a custom parser for "args"
        }
        default {
          # unknown handling
          set m "[current method]: unknown sub-method '$sub'. Available: sub1 sub2"
          return -code error $m
        }
      }
    }
}

o foo sub1 arg1 arg2; # OK
o foo sub2 -np1 arg1 -np2 arg2 arg3; # OK
o foo sub3; # --> foo: unknown ensemble method 'sub3'. Available: sub1 sub2
```

While this switch-threaded method implementation certainly mimics sub-commands (i.e., `foo sub1` and `foo sub2`) to a certain extent, there are considerable limitations, potentially affecting code evolution and maintenance tasks:

1. *Homogeneous vs. Heterogeneous Signatures*: To begin with, there is a tension between providing heterogeneous signatures for ensemble methods and reusing the built-in parameter processing infrastructure. In the above example, the intention is to constrain `foo sub1` to requiring two positional parameters only, while `foo sub2` accepts two non-positional parameters. The parent method `foo` effectively shares its method parameter specification with its children, with the variable argument vector (`args`) not enforcing any further parameter constraints on behalf of the ensemble methods. This leaves the developer with the only option to enforce the signature constraints specific to each ensemble method in the respective switch branch by providing for custom argument parsing.
2. *Blinded introspection*: The built-in object introspection is not aware of the very existence of the ensemble methods nested under `foo`, nor their possibly deviating method parameter specifications. For example, `o info methods foo` and `o info callable methods` won't reveal the two ensemble methods `foo sub1` and `foo sub2`. As one of the consequences, introspection cannot be leveraged to implement ensemble methods. In the above example, the list of available ensemble methods must be maintained explicitly for generating the unknown error message.

3. *Nesting level limitations*: Any implementation variant based on conditional control structures (e.g., switch threading) risks adding further complexity with each further nesting level added to an ensemble method hierarchy (e.g., `foo sub1 sub4`). As each nesting level turns into a nested conditional, e.g., scattered across several switch threads in the example above, the implementation suffers from extra complexity due to dealing with parameter specifications and unknown handling for ensemble methods.
4. *Unknown handling*: The built-in unknown handling of NX is an important meta-programming vehicle. The native unknown handling is sidetracked by the requirement for the switch-local unknown handling. That is, the `default` switch branch replaces the otherwise responsible `unknown` method for objects. Also, unknown handling must be implemented for each and every method ensemble repeatedly; unless facilitated by a piece of meta-programming. Adding nesting levels further complicates this form of ensemble-specific unknown handling.
5. *Method combination*: Combining ensemble methods with refining ensemble methods provided by intrinsic (superclasses) or by extrinsic classification (mixin classes) is hindered. First, the scope for combining methods is the parent selector only. In our example, refining methods can only hook onto the selector `foo`, without further specifying an ensemble method as its refinement target. Second, using `next` chaining in a linearized order of refining `foo` methods becomes non-obvious and error-prone as the scope of `next` calls is the top-level method only.
6. *Method reuse*: The type and the implementation of ensemble methods cannot be reused. This is, to a large extent, due to the limitations of method combination (see the previous item). However, ensemble method implementations based on conditionals are also not accessible to other composition techniques, most importantly method aliases.

Besides, the effects of excessive tangling throughout conditional blocks (e.g., the "Switch Statement Smell" in [10]) and the non-orthogonal extensibility for method ensembles are the consequences. To overcome these limitations, NX supports ensemble methods natively. Ensemble methods are implemented by an advanced form of object delegation hierarchies. A variant of method objects [10], referred to as ensemble objects, are recorded as methods with a registration object. In the above example, `o` acts as the registration object for an ensemble object `foo`, so that `foo` becomes dispatchable as the method member `o foo`. To avoid common pitfalls of method objects, in particular self schizophrenia, special dispatch semantics apply: First, exclusively per-object methods of the ensemble objects provide the leaf methods in a method ensemble hierarchy. Second, the dispatch to an ensemble method is bound to the self-object context of the registration object. With some syntactic sugar, which

effectively hides the declaration ensemble objects and the building of their delegation hierarchies, NX allows one to rewrite the example from Listing 7 as:

Listing 8: Definition of Ensemble Methods with Language Support

```
Object create o {
  :public method "foo sub1" {p1 p2} {
    # ...
  }
  :public method "foo sub2" {-np1 -np2 p3} {
    # ...
  }
}

o foo sub1 arg1 arg2; # OK
o foo sub2 -np1 arg1 -np2 arg2 arg3; # OK
o foo sub3;
# --> Unable to dispatch sub-method "sub3" of ::o foo; valid are: foo sub1, foo sub2
```

Such method ensembles can be incrementally extended, indirected by mixins and filters, and easily shared between objects through method aliasing. To complete the support for ensemble methods, object introspection is fully aware of ensemble methods. One can resolve the entire method path, for which a given ensemble method is registered, from within the ensemble method (via `nx::current methodpath`). Also, introspection makes the unfolded method paths available for querying by method path patterns (using e.g. `/obj/ info methods ?-path? … ?pattern?`).

Public, Protected, and Private - Module Encapsulation versus Method Combination

A primary reason for putting units of code (i.e., object, classes) into relation (e.g., instance-of, superclass/subclass) is to establish various kinds of reuse between these code-units. These relations establish ways of accessing, using, or mutating structural and behavioral features (primarily instance variables and methods) of these units. For example, by method combination (using the `next` primitive) a subclass may use the methods of its superclasses. A similar reuse can be achieved by mixin classes, by traits or, at the method level, by method aliases and method forwards.

When reusing complex units of code (e.g. deep class hierarchies), which have possibly been developed by different teams and which have been constantly refactored, one danger arises from unwanted interactions, such as the accidental shadowing of methods. The example of a constructor anomaly given earlier falls into this category of unwanted interactions.

The more relations between code-units are established and the more bloated object interfaces become, the more likely unwanted interactions will occur. To manage such

interactions, it is important to define explicit and strict module interfaces [14]. The literature employs the notion of *module encapsulation* for describing means to regulate the accessibility, the use, and the changeability of module features by other modules [13].

NX supports stronger means for module encapsulation than XOTcl. The design goal of NX was to encourage encapsulation by language constructs rather than prohibiting access at all. For example, denying any access to an object's state would make serialization of objects from the scripting language impossible since the serializer needs access to all internals. NX adds the following means of module encapsulation:

1. In NX, the object state (instance variables) is better protected than in XOTcl by not providing any publicly available, built-in accessor methods to all instance variables. XOTcl, on the contrary, exposed the methods `set` and `unset`; or, the general variable importer `instvar`. The *access to instance variables* from within instance methods is encouraged in NX via Tcl's variable resolvers and the colon prefix.
2. The redefinition of behavioral object features (in particular methods and properties) can be restricted by declaring the object features *redefine-protected*.
3. NX provides a fine-grained mechanism to establish method *call-protection* between objects and classes. An object can expose three different method sets at the same time:
    a. The `public` method set is usable by any client object, without restrictions. The methods of this set can be targeted by self-calls (e.g., `:bar` in the example below), next-calls, and command-calls (i.e., when specifying the object's Tcl command name as receiver: `a1 foo`).
    b. The `protected` set restricts the method's use to self-calls and next-calls. That is, calling upon the method set through the command reference of the object is forbidden.
    c. The `private` interface is restricted to self-calls and to call sites defined for the same class or object scope as the called method.

```
nx::Class create A {
  #
  # Public interface of class "A"
  #
  :public method foo args {
    :bar              ; # invoke protected method of current object
  }

  #
  # Protected interface of class "A"
  #
```

```
  :protected method bar {} {
    : -local baz     ; # invoke private method of current object with "-local" flag
    # ...
  }

  #
  # Private interface of class "A"
  #
  :private method baz {} {
    # ...
  }
}
A create ::a1
```

In the above listing, the method modifiers `public`, `protected`, and `private` are used to add methods to these three method sets. If omitted, the default call protection in NX is `protected`. This default can be altered by configuration. From within methods of the instance `::a1`, the protected and the private method sets can be effectively used. The method call statement `:bar` represents a self-call to the protected method set. The invocation of a private method is performed via `: -local baz`. The flag `-local` indicates to call only methods from the private method set. The flag `-local` at the call site makes the intention clear to use only a method declared for the same class context. It cannot be invoked from within methods of subclasses (as the following example shows), nor from methods of superclasses.

However, when the methods `foo`, `bar`, and `baz` are called from the "outside" (i.e., from instances of other classes, or from the top-level namespace), neither the protected, nor the private methods of `A` are callable:

```
a1 foo; # command-call to public interface --> OK
a1 bar; # command-call to protected interface --> ::a1: unable to dispatch method 'bar'
a1 baz; # command-call to private interface --> ::a1: unable to dispatch method 'baz'
```

Let us now introduce a superclass/subclass relation between the classes `A` and `B`, with the sublcass `B` defining its own public method set consisting of the methods `bar` and `baz`:

```
nx::Class create B -superclass A {
  :public method bar {} {
    next    ; # next-call to protected interface --> OK
  }
  :public method baz {} {
    next    ; # next-call does not reach the private method
  }
}

B create ::b1
b1 bar; # command-call to public interface --> OK
```

The public method `B bar` shadows `A bar`. Because `B bar` can be called unrestrictedly, it can be invoked from the outside. Since protected methods are available for next-calls, `A bar` can be reused via `next` in this context.

The method `B baz` is part of the public interface of class `B` and defines a next-call. While `A baz` is a candidate target for this next-call, however, since private methods are not available to next-calls, the invocation of `next` behaves exactly like `A baz` would not have been defined.

The redefinition protection and the call protection in NX are implemented by a set of properties assignable to method implementations through NSF primitives. Based on these property assignments, the language runtime regulates the modification of the method implementations (redefinition protection) and determines the availability of method implementations as message receivers depending on the caller context (call protection). This low-level interface allows the NSF language developer to specify custom redefinition and call protection schemes. For example, for XOTcl 2.0, the default call protection mode is so implemented as `public`.

To summarize, discriminating between `public` and `protected` methods provides for defining explicit object interfaces (i.e., intended ways of having classes and objects reused by client objects). The `private` modifier helps hide implementation details and helps avoid unwanted method combinations due to name clashes in, e.g., mixin classes or traits.

Support for Advanced Feature Composability: Traits

NX supports the concept of per-object, per-class, and transitive per-class mixins [7]. In addition to mixins, NX adds a variant of traits [6] as a scripted language extension. Traits realize a composition mechanism for the reuse of methods. Contrary to other forms of reuse (e.g. inheritance of methods in a class hierarchy or via mixin classes), the methods defined in traits are materialized in the target objects and classes. For the implementation of the traits, method aliases provide the necessary implementation infrastructure. Every method inherited from a trait can be modified, deleted etc. by subsequent method definitions for a given class. This gives more fine-grained control over the reuse of methods and overcomes the "total composition ordering" limitation of mixins [6]. Consider the following example of a simple trait called `tReadStream` which provides the interface to a stream:

```
package require nx::trait

nx::Trait create tReadStream {
  #
  # Define the methods provided by this trait:
```

```
  #
  :public method atStart {} {expr {[:position] == [:minPosition]}}
  :public method atEnd {} {expr {[:position] == [:maxPosition]}}
  :public method setToStart {} {set :position [:minPosition]}
  :public method setToEnd {} {set :position [:maxPosition]}
  :public method maxPosition {} {llength ${:collection}}
  :public method on {collection} {set :collection $collection; :setToStart}
  :public method next {} {
    if {[:atEnd]} {return ""} else {
      set r [lindex ${:collection} ${:position}]
      :nextPosition
      return $r
    }
  }
  :public method minPosition {} {return 0}
  :public method nextPosition {} {incr :position 1}

  # This trait requires a method "position" and a variable
  # "collection" from the base class. The definition of the trait is
  # incomplete in these regards.
  :requiredMethods position
  :requiredVariables collection
}
```

Define a class `ReadStream` with properties `position` and `collection` which uses the trait. The method `require trait` checks the requirements of the trait and imports the methods of the trait into the class `ReadStream`:

```
nx::Class create ReadStream {
  :property {collection ""}
  :property {position 0}
  :require trait tReadStream
}
```

One can now create an instance of the class `ReadStream` ...

```
ReadStream create r1 -collection {a b c d e}
```

to test the behavior of the composed class:

```
% r1 atStart
1
% r1 atEnd
0
% r1 next
a
% r1 next
b
```

NX supports simple and composite traits, with a composite trait definition inheriting from another trait.

MongoDB Mapping

The NSF development toolkit features a Tcl/C-API generator and Tcl_Obj type converters for developing NSF/C extensions. By using these helpers, we developed a MongoDB binding for NX. The C-implemented part of this extension integrates with the C client library of MongoDB. The extension also provides an NX/Tcl package for integration of NX objects with MongoDB.

The MongoDB extension provides both a low-level interface and a high-level, object-oriented interface based on NX. By using this high-level API, one can create NX classes and objects which are equipped with additional capabilities for defining (`property`, `index`), retrieving (`find`), and storing (`save`) objects in MongoDB. The example below shows an excerpt from the *"Business Insider"* data model, a frequently cited MongoDB showcase [16]. The listing depicts the entity definitions for postings, authors, comments, tags etc. Using the parameter option `embedded`, one can created embedded (nested) documents with the required multiplicity. In this example, we also use the `incremental` setter interface for creating tags.

```
package require nx::mongo

nx::mongo::db connect -db "tutorial"
#
# Create the application classes based on the "Business Insider" data
# model. Note that instances of the class "Comment" can be embedded in
# a posting (property "comments") as well as in a "Comment" itself
# (property "replies"). All comments in this example are multivalued
# and declared "incremental" (i.e., one can use slot methods "... add
# ..." and "... delete ..." to add/remove values of the multivalued
# attributes).
#
nx::mongo::Class create Comment {
  :property author:required
  :property comment:required
  :property replies:embedded,incremental,type=::Comment,0..n
}

nx::mongo::Class create Posting {
  :index tags
  :property title:required
  :property author:required
  :property ts:required
  :property comments:embedded,incremental,type=::Comment,0..n
  :property tags:incremental,0..n
}

# Create a Posting
set p [Posting new -title "Too Big to Fail" -author "John S." \
          -ts "05-Nov-09 10:33" -tags {finance economy} \
```

```
          -comments [list \
               [Comment new -author "Ian White" -comment "Great Article!"] \
               [Comment new -author "Joe Smith" -comment "But how fast is it?" \
                 -replies [list [Comment new -author "Jane Smith" -comment "scalable?"]]] \
               ]]

# We add an additional comment at the end of the list of the comments
# using the incremental operation "add" ...
$p comments add [Comment new -author "Gustaf N" -comment "This sounds pretty cool"] end

# ... and we add yet another tag ...
$p tags add nx

# ... and save everything
$p save

# Now fetch the first entry with the tag "nx"
set q [Posting find first -cond {tags = nx}]
....
```

Infrastructure and Toolkit

For developing object systems and programs in NSF/NX, a rich development environment is available. Monitoring the runtime performance is possible through a DTrace binding and a native measurement facility. Detecting skewed refcounts is facilitated by a built-in monitoring facility for Tcl_Objs which complements Tcl's `memory` command. For defining Tcl/C APIs based on the uniform parametrization infrastructure of NSF, an API generator based on a declarative API specification language can be used. Functional tests can be managed using the `nx::test` environment, a documentation generator (`nxdoc`) takes Javadoc-styled Tcl comment blocks as input and outputs to various templating backends (e.g., YUIDoc markup documents or wiki pages).

The following listing shows a D script for DTrace which turns DTrace probes on and off during a script run. The D script measures (when activated) the time spent in methods called on `nx::Object`. Finally, it provides a graph produced by the DTrace `quantize` aggregator function.

```
/* -*- D -*-
 *
 * Quantize time between method-entry and method-returns for calls on nx::Object
 *
 * Activate tracing between
 *    nsf::configure dtrace on
 * and
 *    nsf::configure dtrace off
 *
 */

nsf*:::configure-probe /!self->tracing && copyinstr(arg0) == "dtrace" / {
```

```
  self->tracing = (arg1 && copyinstr(arg1) == "on") ? 1 : 0;
}

nsf*:::configure-probe /self->tracing && copyinstr(arg0) == "dtrace" / {
  self->tracing = (arg1 && copyinstr(arg1) == "off") ? 0 : 1;
}

/*
 * Measure time differences on method calls on nx::Object
 */
nsf*:::method-entry /self->tracing && copyinstr(arg1) == "::nx::Object"/ {
  self->start = timestamp;
}

nsf*:::method-return /self->tracing && copyinstr(arg1) == "::nx::Object" && self->start/ {
  @[copyinstr(arg1), copyinstr(arg2)] = quantize(timestamp - self->start);
  self->start = 0;
}
```

The snippet below shows how DTrace can be applied to monitor the evaluation of a NSF/NX test script, as well as how the result is rendered (showing here just a small part of the output). The NSF distribution contains some more examples for using DTrace with NSF/NX.

```
% sudo TCLLIBPATH=. dtrace -F -s dtrace/timestamps-q.d -c "./nxsh tests/object-system.test"
....
  ::nx::Object                                              eval
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@@                          1
           65536 |                                         0

  ::nx::Object                                              vars
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 5
           16384 |                                         0
```

A Synthetic Performance Evaluation

This section presents a first performance comparison between NX and XOTcl 2.0, on the one hand, and XOTcl 1.6.0, on the other hand. The measurement design is comparable to the one presented in [3]. The data for NX, XOTcl 2.0, and XOTcl 1.6.0 were gathered running on top of the same Tcl versions (especially Tcl 8.5.10 and Tcl 8.6b2) and using the same machine (3.33 GHz Intel Core 2 Duo) under Mac OS X 10.6.8. All C-programs and Tcl libraries were compiled with gcc 4.2.1 and identical compiler flags (in particular, `-O3`).

The first probes used to gather execution times were adopted from the methcall benchmark of the OO shootout (http://wiki.tcl.tk/2428). By doing so, the results can be related to previously published benchmark reports for other Tcl object systems.

In addition, probes for object creation and object deletion times are included. To be precise, we measured the average execution time to create and to destroy a single object while creating/destroying 100.000 objects.

Table 3. Comparison of the OO Shootout Benchmark, Object Creation and Deletion

	NX 2.0 Tcl 8.5.10	NX 2.0 Tcl 8.6b2	XOTcl 2.0 Tcl 8.5.10	XOTcl 2.0 Tcl 8.6b2	XOTcl 1.6.0 Tcl 8.5.10	TclOO 0.6 Tcl 8.5.10	TclOO 0.6.3 Tcl 8.6b2
OO Shootout: methcall (n=30.000)	0.57	0.79	2.07	2.78	2.61	1.40	2.07
Object create (n=100.000)	35.63	40.19	39.71	45.46	56.77	48.56	49.85
Object destroy (n=100.000)	20.95	23.21	20.97	22.89	25.77	269.70	257.98

The first table row gives the OO shootout methcall timings. It reports the average timing for 30.000 iterations of the method call probe. The second and third rows show the timings for object creation and deletion. The timing measure is the average execution time per operation in micro seconds (hence, smaller values indicate a better performance).

Figure 9 visualizes the measurement provided in Table 3 in terms of performance improvements relative to XOTcl 1.6.0 (index: 100). The higher the indices, the more substantial is the relative improvement. The chart shows that NX is 4.5 times faster than XOTcl 1.6.0 for the OO Shootout methcall probe, both running Tcl 8.5.10. The methcall performance of NX 2.0 under Tcl 8.6b2 slightly decreases. Despite this, both NX probes give the best result. The methcall script used for XOTcl 1.6.0 and for XOTcl 2.0 are the same (i.e., the new language features of XOTcl 2.0 are not used). For Tcl 8.5.10, XOTcl 2.0 is about 26% faster than XOTcl 1.6.0. The object creation and object deletion probes draw a similar picture. NX is the fastest under Tcl 8.5.10. TclOO appears to be especially slow on destroying objects, both under Tcl 8.5.10 and Tcl 8.6b2.

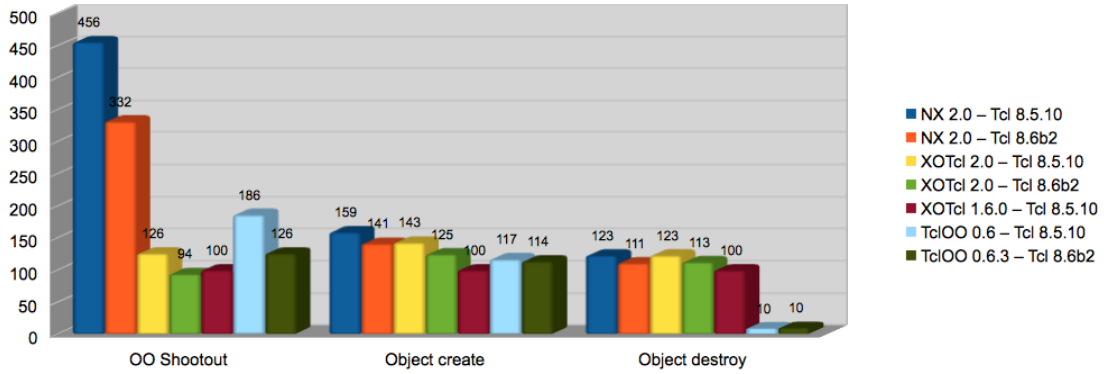

Figure 9. Performance Improvements (relative to XOTcl 1.6.0) on the OO Shootout Benchmark, Object Creation and Deletion.

The second set of measurement probes aims at capturing the execution timings of method dispatches for different parameter handling and argument parsing tasks. The method implementations used as probes have trivial bodies (no-ops might be treated differently by the byte-code compiler). The first probe, `args0`, is a method without parameters. The method `args3` specifies three positional parameters, `np2` expects two non-positional parameters and `np2args3` has two non-positional and three positional parameters. None of the parameter specifications in these probes contains parameter value constraints, which would have to be scripted in XOTcl 1.6.0 and TclOO, inducing a considerable performance penalty.

```
nx::Class create C {
  :public method args0 {} {return 1}
  :public method args3 {x y z} {return $x}
  :public method np2 {{-a 10} {-b 100}} {return $a}
  :public method np2args3 {{-a 10} {-b 100} x y z} {return $x}
}
#
# Measuring the following method invocation on instance "c1" of class "C":
#    c1 args0
#    c1 args3 1 2 3
#    c1 np2
#    c1 np2args3 -a 20 -b 200 1 2 3
```

Table 4 presents the collected probe throughput in terms of calls per seconds (higher numbers are better). The results are illustrated as a chart in Figure 7. XOTcl 2.0 is not reported separately in this test since it builds upon the same parameter/argument handling infrastructure as NX. The NX timings apply to XOTcl 2.0. Also, the comparison for non-positional parameter handling does not cover TclOO, since it does not feature a built-in implementation for non-positional parameters. A pure Tcl implementation would be substantially slower.

Table 4. Calls per Seconds on Method Dispatches

	NX 2.0 Tcl 8.5.10	NX 2.0 Tcl 8.6b2	XOTcl 1.6.0 Tcl 8.5.10	TclOO 0.6 Tcl 8.5.10	TclOO 0.6.3 Tcl 8.6b2
args0	3,074,463	2,113,561	1,360,003	2,499,743	1,942,890
args3	2,609,651	1,815,349	1,175,925	2,069,303	1,711,060
np2	2,198,836	1,553,550	481,428	n.a.	n.a.
np2args3	1,440,079	1,116,283	250,525	n.a.	n.a.

Figure 10 provides a graph with values of Table 3 illustrating the performance index against XOTcl 1.6.0 (which has for every test a performance index of 100). These tests show that especially for non-positional argument handling NX improves substantially over XOTcl 1.6.0, by factors of up to 5.75. NX shows the best performance profile for all parameter handling tests. Similar to the methcall probes above, when NSF is compiled against Tcl 8.6b2, the parameter handling performance degrades significantly as compared to the same NSF version built against Tcl 8.5.10.

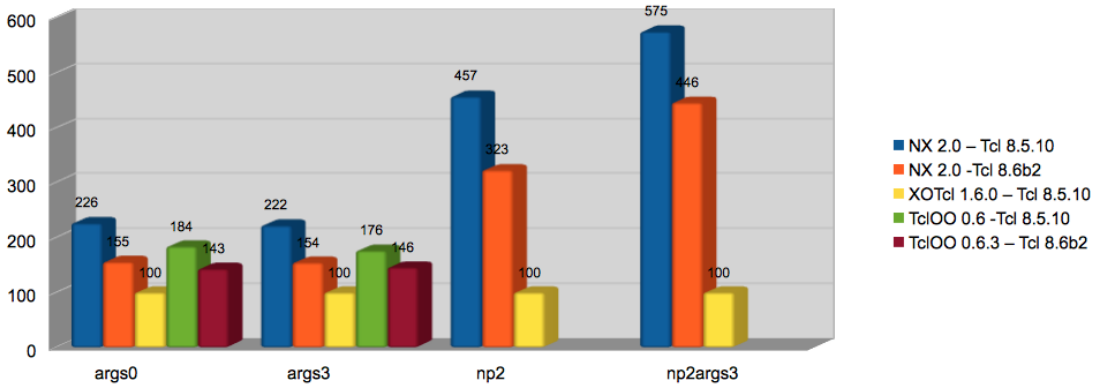

Figure 10. Performance Improvements on Method Dispatches (as compared to XOTcl 1.6.0)

Summary and Availability

For this paper, we were motivated to present a comprehensive overview of the features of the Next Scripting Toolkit, and the Next Scripting Language (NX) in particular. We gave a first insight into advancements for the NX concrete syntax (i.e., init blocks and the colon prefix) and discussed the basics of object and method parameters. The overview was completed by walking the reader through the enhancements for defining behavioral features of objects, i.e., method aliasing, method ensembles,

and method call protection. The interplay of these features was demonstrated by introducing NX traits as an important composition technique. We concluded by hinting at developer support tools (e.g., DTrace) and at first libraries realized for NX, most importantly a MongoDB binding for NX.

NSF, NX and XOTcl 2.0 will become publicly available at the time of the Tcl/Tk 2011 Conference from http://next-scripting.org/.

References

- [1] Bracha G. (2004): Pluggable Type Systems. In Proceedings of the OOPSLA'04 Workshop on Revival of Dynamic Languages (RDL 2004).
- [2] Clarke S., Becker C. (2003): Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In Proceedings of the 15h Workshop of the Psychology of Programming Interest Group (PPIG 2003), Keele, UK (pp. 359—336).
- [3] Neumann G., Sobernig S. (2009): XOTcl 2.0 — A Ten-Year Retrospective and Outlook. In Proceedings of the Sixteenth Annual Tcl/Tk Conference, Portland, Oregon, 2009. Tcl Association.
- [4] Neumann G., Zdun U. (2000): XOTcl — An Object-Oriented Scripting Language. In Proceedings of the 7th USENIX Tcl/Tk Conference (cl2k), Austin, TX, USA, 2000.
- [5] Fellows D.K. et al. (2008): Object Orientation for Tcl. TIP#257, finalized in September 2008. URL http://www.tcl.tk/cgi-bin/tct/tip/257.html.
- [6] Ducasse S., Nierstrasz O., Schärli S., Wuyts R. , Black A. P. (2006): Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2): 331-388 (2006).
- [7] Zdun U., Strembeck M., Neumann G. (2007): Object-Based and Class-Based Composition of Transitive Mixins, Information and Software Technology, 49(8) 2007.
- [8] Fowler M. (2009). Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, last accessed: July 7, 2009, 2005
- [9] Cohen T., Gil, J. (2007): Better Construction with Factories. Journal of Object Technology, 6(6), 103—123.
- [10] Fowler, M. (2003): Refactoring - Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

- [11] Renner P., Rauschmayer A. (2005): TUBE - Structure-Orientation in a Prototype-Based Programming Environment. In Proceedings of the 2005 International Conference on Programming Languages and Compilers, PLC 2005, Las Vegas, Nevada, USA, June 27-30, 2005 (pp. 194-200). CSREA Press.
- [12] Fähndrich M., Leino, K. R. M. (2003): Declaring and Checking Non-null Types in an Object-Oriented Language. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2003), Anaheim, California, USA, New York, NY, USA, 2003 (pp. 302-312). ACM.
- [13] Schärli N., Black A. P., Ducasse S. (2004): Object-oriented Encapsulation for Dynamically Typed Languages. In Proceedings of the OOPSLA'04. ACM.
- [14] Buschmann, F. & Henney, K. (2003). Explicit Interface. In Proceedings of EuroPLoP 2003, Irsee, Germany, 2003.
- [15] Sobernig, S., Gaubatz, P., Strembeck, M., & Zdun, U. (2011). Comparing Complexity of API Designs: An Exploratory Experiment on DSL-based Framework Integration. In Proceedings of the 10th International Conference on Generative Programming and Component Engineering (GPCE'11), Portland, OR, USA, 2011.
- [16] White, I. (2009). How This Web Site Uses MongoDB, URL: http://www.businessinsider.com/how-we-use-mongodb-2009-11, last accessed: October 8, 2011.