Towards XOTcl 2.x[*]
A Ten-Year Retrospective and Outlook

Gustaf Neumann, Stefan Sobernig

Institute for Information Systems and New Media

Vienna University of Economics and Business (WU Vienna), Austria

{gustaf.neumann|stefan.sobernig}@wu.ac.at

Abstract

Recent work on the Extended Object Tcl (XOTcl) was geared towards the orthogonality, the ease of use, the productiveness, and the tailorability of the language. The result is an innovative object-oriented language framework which serves for developing a family of object-oriented Tcl dialects. In this work-in-progress report, we map the background and history of advanced language constructs (i.e., mixin classes, filters, method delegation) and their continued refinement (i.e., transitive mixins, mixin and filter guards). We present the infrastructure for creating derivative Tcl OO dialects (i.e., creating object systems and their structural relations, assembling base object behavior). A canonical model and infrastructure of parametrization of commands, methods, and objects is presented. Important steps of internal re-designing and refactoring (callstack and object life-time management) are discussed. Execution time and call throughput measurements for basic object life-time and method dispatch scenarios are reported, exhibiting substantial improvements over the XOTcl 1.6.x branch and TclOO 0.6.

1 Introduction

In this paper, we review the continued development of the Extended Object Tcl (XOTcl) language and programming framework. We demonstrate major feature additions and enhancements since the initial XOTcl presentation in early 2000 [17]. Against this background, we shall build up an overview of future directions in the ongoing development of a revised XOTcl 2.x infrastructure. XOTcl 2.x targets language-oriented programming by providing a framework for developing derivative object-oriented Tcl dialects and embedded, domain-specific languages. First, however, we shall briefly recall XOTcl's history which continuously led to developing XOTcl 2.x.

The 1.x branch of the Extended Object Tcl (XOTcl) language was originally designed to provide language support for realizing object-oriented design patterns [7] in the Tcl scripting language

[*]published as: Gustaf Neumann and Stefan Sobernig. XOTcl 2.0 – A Ten-Year Retrospective and Outlook. In *Proceedings of the Sixteenth Annual Tcl/Tk Conference*, Portland, Oregon, October 2009. Tcl Association. URL http://nm.wu-wien.ac.at/research/publications/b806.pdf

[9]. The language represented an important contribution, both as an academic and as a language artifact. Consequently, to this date, XOTcl is documented in more than 20 publications (see [25] for an overview).

The object system of the Extended Object Tcl (XOTcl) has two properties, different from many other object-oriented languages: (1) it is object-centric (rather than class-centric such as C++, Java; see [24]) and (2) it supports dynamic relationships between objects. In class-centric object systems, the properties of objects are solely defined by the classes. Therefore, in class-centric approaches, the relation between a class and an instance of this class is immutable. In an object-centric approach, all information about the object is contained in the object itself (e.g., its variables). Objects can have object-specific properties which are not defined by any class of which the object is member (e.g., object-specific methods). Objects can have as well relations to classes which act as object factories and method repositories.

In XOTcl's object-centric approach, all relations between objects and classes or between classes themselves can be changed at any time, for instance, to reflect changes in object roles (e.g., students become employees) or to add/remove behavior to single or multiple objects dynamically (e.g., add logging, visualizations of structural object relations, etc.). All kinds of object relations can be altered. For example, one can change superclass relations of classes or re-class objects. In addition, member definitions for objects and classes are mutable at runtime so that methods and variables can be added or removed dynamically.

XOTcl delivers advanced object-oriented abstractions such as mixin classes. These abstractions were incorporated in response to the analysis of common abstraction mismatches observed for object-oriented designs [9]. To overcome such mismatches, the language supports instantiating design patterns [7] based on a minimal set of language constructs for object composition and message interception, i.e., filters [9] and mixin classes [15, 16, 28]. XOTcl was one of the first languages providing this kind of language-level support for design patterns.

The set of language features initially presented in February 2000 [17] was conceptually complete and so entered the major release at the root of the 1.x branch in November 2002. This original feature set included models of multiple class-based and mixin-based inheritance (i.e., per-object and per-class mixin classes), filters as a message interception technique, and a flexible scheme of object and class aggregation through the integration with Tcl namespaces. In addition, constructs to express constraints over object-type behavior were provided, i.e., invariants as well as pre- and post-conditions. Since then, the XOTcl 1.x language has seen a considerable number of feature additions and enhancements. These were motivated by incorporating findings of our ongoing software engineering research and by the experiences gathered in developing XOTcl-based applications.

The remainder of this paper is organized as follows. To begin with, we trace the history of major language concepts and their further-development in Section 2. This spawns the general background for our work on XOTcl 2.x which is sketched in the two follow-up sections: Section 3 introduces the reader to novel language features and feature consolidations achieved for XOTcl 2.x. Section 4, on the contrary, maps XOTcl 2.x implementation internals and major refactorings. Finally, in Section 5, we report on a first, tentative performance evaluation obtained from profiling essentials such as object generation and method dispatches. We conclude by summarizing on lessons learned and on remaining challenges (see Section 6).

2 The Evolution of XOTcl 1.x

Concept-wise, XOTcl's mixin classes were refined to realize the concept of transitive mixins [28]. Transitive mixins provide for the orthogonal refinement *between* mixin-based extension components, rather than the mixin-based extension of base components alone. With transitive mixins, it is possible to mix-in class trees into target objects and target classes explicitly, as well as to apply to the targets implicitly the extension behavior mixed into the mixin-classes themselves (i.e., multiple layers of mixins are applied *by transition*). This increases orthogonality and facilitates mixin-class management. A further improvement was the introduction of predicate expressions (guards) for the conditional dispatch of filters and for the conditional injection of mixin classes. This allows for modeling much more detailed interaction semantics between objects and classes, as suggested by the subject-oriented programming approach (see, e.g., [19]).

Another important conceptual addition are slots (starting with XOTcl 1.5). Slots are mediator objects owned by classes which provide means to define instance fields and extensible protocols for interacting with them. By extensible, we mean that object field behavior can be specified which goes beyond the fixed accessor/mutator protocol. Also, means of method delegation were added (i.e., per-object and per-class forwards). Moreover, non-positional method parameters [26] including extensible type predicates (the so-called *check options*) were realized.

The XOTcl 1.x language and its infrastructure components have turned into vehicles for research- as well as production-grade application development. In the context of the web application framework OpenACS [18], XOTcl-based framework extensions (in particular an OO layer on top of the procedural core infrastructure; see [11]) and application modules were developed. Examples include the versatile wiki framework XOWiki [12] and the object remoting bundle xorb [20]. Beyond OpenACS, the web framework ActiWeb [17], and the service-oriented, peer-based middleware framework Leela [27] are written in XOTcl. For realizing role-based access control (RBAC) and RBAC-specific role engineering, XOTcl was chosen as the programming infrastructure for the toolkits xoRBAC [14] and xoRET [21], respectively. More recently, the XOTcl-based framework POKER [22] was presented which assists at realizing extensible, event-based policy infrastructures.

XOTcl has reached a much wider audience through industry adoption: Archiware [1] offers high-performance backup and restore products using XOTcl-based components. Cisco has been reported to have adopted XOTcl along with Tcl for embedded systems running its router and firewall products. More recently, Cisco sponsored the development of an XOTcl plug-in for the Eclipse Dynamic Language Toolkit (DLTK; [3]). Finally, XOTcl ships with major OS distributions and development packs, such as Mac OS X 10.4 (Tiger) and Debian Lenny.

Last but not least, XOTcl served as the conceptual basis for TclOO, defined in TIP#257 [5]. After reviewing the existing Tcl extensions for object orientation, many of the XOTcl concepts (in particular, mixin classes, method delegation, filters) were eventually judged to be suitable for entering the core OO infrastructure TclOO (see, e.g., [4, 5]), which is included in the forthcoming release of Tcl 8.6. TclOO is, intentionally, a non-compatible subset of XOTcl 1.x.

3 Beyond XOTcl 1.x

During developing and discussing TIP#257 [5], a major issue arose regarding the primary objective of a core OO extension to Tcl: Was it to be designed as *the* long-awaited OO dialect for Tcl? Or, was it to serve for hosting multiple, existing Tcl OO dialects? In response to this thread, the TIP#279 [10] proposed a framework for building Tcl OO dialects. Based on this TIP and our reference implementation, we commenced the parallel development of an XOTcl 2.x branch, aiming at language-oriented programming [6] for Tcl and beyond.

The modified XOTcl 2.x language sets out to consolidate the range of existing features and, at the same time, to generalize XOTcl as a language for deriving and hosting object-oriented Tcl variants. In the following sections, we provide a practical overview of the forthcoming XOTcl 2.x release. We will focus on the framework aspects to support a family of object-oriented languages based on a common set of features. Also, we will introduce selected features improving the orthogonality and extensibility of the XOTcl 2.x language.

3.1 Language-Oriented Programming

General-purpose OO programming languages, such as XOTcl, serve as hosting languages for derived OO languages and embedded domain-specific languages (so-called internal DSLs [6]). In order to engineer derivative languages, it is often necessary to tailor their OO feature set towards the actual needs of some application. By preserving, and eventually only hiding, the general-purpose language feature in the DSL, the host language introduces unwanted complexity and the risk of breaching DSL semantics by the DSL-using developers. Hence, to anticipate such risks, novice DSL-using developers must take a steep learning curve to get acquainted with the entire feature set of the hosting language. From the perspective of the XOTcl 2.x infrastructure, Tcl OO dialects (including the XOTcl 2.x language itself) are examples of such derived languages. Judging from the high number of available Tcl OO dialects, the basic concepts of derived languages seems to be well accepted by the Tcl community.

The XOTcl 1.x language already provided a powerful metalevel programming model (e.g., referred to as "metaobject protocol" in CLOS [8]). It provided runtime access to selected language primitives of the XOTcl 1.x language through the extensive introspection capabilities (e.g., `info` operations) as well as intercession through meta-programming, interception techniques (filters and mixin classes), and meta-classes. However, the metalevel programming capabilities were aligned to the basic XOTcl 1.x language model alone, disallowing access and mutation of certain model elements (e.g., the root level of the object system). In XOTcl 2.x, the emphasis is on opening up not only a single language's set of primitives but also on introducing primitives to specify a language model as such. Derived language models, then, become accessible through *their* own metalevel programming model. Abstractions and language model semantics for derived languages (e.g., the XOTcl 2.x *language*) can directly be specified in this host language (hereafter, the XOTcl 2.x *infrastructure*).

The new XOTcl 2.x infrastructure delivers an OO language framework to assemble different object systems from a minimal core set of entity types, relationship types, predefined methods, gluing Tcl commands, and a shared runtime infrastructure (e.g., a versatile, directly accessible method dispatcher). In the following, we outline the essentials of the XOTcl 2.x infrastructure by giving examples of the XOTcl 2.x language implementation. For a detailed, yet slightly outdated presentation see TIP#279 [10].

The XOTcl 2.x infrastructure is defined based on a set of language-programming primitives which are exposed at the scripting level (in the `xotcl` namespace). Using these primitives, object systems can be specified, e.g., by defining their own basic class hierarchies and method sets. A new object system can be defined via the command `::xotcl::createobjectsystem` (see also Listing 1). This commands takes as parameters the name of the meta-class object used for defining classes and a root class object which describes the common behavior of all objects.

```
::xotcl::createobjectsystem ::xotcl::Object ::xotcl::Class
```

Listing 1: Defining the Root Level of the XOTcl 2.x Language

These base objects (i.e., `::xotcl::Object` and `::xotcl::Class` in Listing 1) do not carry any behavior, i.e., they don't have any predefined methods attached. A language engineer can use methods (in the OO sense) of a predefined (and extensible) method set (e.g., Tcl/C commands) and bind these methods under arbitrary names to the base objects.

```
#
# an XOTcl/C command: Class.alloc()
#
::xotcl::alias ::xotcl::Class alias ::xotcl::cmd::Class::alloc
#
# a Tcl/C command: Object.append()
#
::xotcl::alias ::xotcl::Object append -objscope ::append
```

Listing 2: Assemble the Object Behavior from XOTcl/ Tcl Command Sets

This initial method binding can be achieved for every class/object by using the `::xotcl::alias` command (see also Listing 2). This primitive permits the language developer to register XOTcl/C or Tcl/C commands. This binding does not incur any performance penalty. Single commands and procedures can be bound to several receiver objects, allowing for new forms of reuse and encapsulation. Tcl/C commands can further be bound to the object scope by using the `-objscope` flag so that they operate on the object variables. In addition, the language developer can use method delegation to centralize behavior (i.e., by using `forward`).

```
::xotcl::relation ::xotcl::Class superclass ::xotcl::Object
::xotcl::relation ::xotcl::Object class ::xotcl::Class
::xotcl::relation ::xotcl::Class class ::xotcl::Class
```

Listing 3: Defining Basic Relations of the XOTcl 2.x Object System

So far, the XOTcl 2.x language model describes two base objects with some basic behavior. However, the base objects do not appear to be related at all (apart from exchanging messages, if at all). In a next step, the language designer can define *structural* relations between the base objects (e.g., `class`, `superclass`, `mixin`) by using the `::xotcl::relation` command. In Listing 3, this command is used to lay out the three fundamental relational axes of the XOTcl 2.x object system.[1]

XOTcl 1.x, as well as derived languages based on the XOTcl 2.x infrastructure, appear weakly runtime type-checked. The signature interface of objects, i.e., their object-types based on method signatures as well as the scoped visibility and accessibility of object members, is enforced at invocation time of methods. When designing a derived language core, even this weak form of typing can be bypassed. A generic dispatcher infrastructure (`::xotcl::dispatch`, see also Listing 11) permits the language developer to dispatch predefined, C or Tcl implemented methods on arbitrary objects, regardless of whether they appear available or accessible along the precedence path or not. This mechanism can be used for example for serialization or other introspection purposes.

```
#
# Perform a call to "::Foo.exists(bar)" without a prior method registration
#
# ::xotcl::dispatch <object|class> <command|method> ?parameters ...?

::xotcl::dispatch ::MyObject ::xotcl::cmd::Object::exists bar
```

Listing 4: Objects receiving "alien messages"

This language programming infrastructure provides extremely flexible instruments to create Tcl OO dialects as well as custom language constructs for application frameworks and internal DSLs. The XOTcl 2.x framework hosts as well a language mostly compatible with XOTcl 1.x, built solely using these primitives.

3.2 Objects as Methods and Submethods

Orthogonal, procedural refinements in Tcl are typically realized through wrapper procedures which rename the original command and provide a new command with the same name. Alternatively, commands may be shielded by an explicit dispatch procedure, using `interp hide` and `interp invokehidden`. The new or shadowing procedure might or might not delegate to the renamed or hidden command.

In XOTcl, object behavior is made extensible through method combination, fusing inherited and local versions of a method. One can shadow methods by defined same named methods and delegate to the shadowed methods via `next`. Shadowing of methods can be achieved either through subclassing, mixin classes, or filters. However, in XOTcl 1.x, it is not possible to overload, extend or shadow *sub*commands of methods (like subcommands for `info` or `string`). Consider the example of the XOTcl `info` method used for introspection: The command `o1 info class` returns the class of the object `o1`. While the `info` method can be easily shadowed through method combination, e.g., a mixin class defining an `info` method, the `class` subcommand cannot (in XOTcl 1.x).

To overcome this limitation, XOTcl 2.x provides for language-programming primitives (e.g., `::xotcl::alias`), object aggregation, and method delegation to dispatch objects like methods. This means that the object name can be used as a method, and the methods of this object appear

[1]Note, that the `::xotcl::createobjectsystem` implicitly sets this triad relation for the previously created base objects.

as submethods. There are two implementation variants available to expose objects as methods or submethods: method delegation (i.e., `forward`) and nesting objects. Based on method delegation, introspection was redesigned in XOTcl 2.x using objects (i.e., `::xotcl::objectInfo` and `::xotcl::classInfo`). They are registered with `::xotcl::Object` and `::xotcl::Class`, respectively, under the name `info`. These `info` objects own methods like `class`, `vars`, etc. which, in turn, appear as submethods of the `info` method. By applying mixin classes or filters on the `info` objects, the `info` submethods can be refined. Consider an advanced scenario. An application must stream the states of its objects into a string notation to transfer objects and their states between processes or threads through message passing. In such a scenario, the application can devise a mixin extension `Serializer` which wraps around relevant methods of the `info` objects, in particular `vars`, to introspect on object variables and to produce a script fragment. This script is then evaluated in the new execution context (i.e., the target interpreter) to replicate an object state (i.e., its object variables and their current values).

Apart from method delegation, object aggregation can be used to obtain refineable submethods. This technique is, for instance, used in the slots [25] infrastructure in XOTcl 2.x. Nested objects are exposed as methods owned by their parent objects. This allows to dispatch method calls to the nested objects in a notation closely resembling the subcommand convention for standard Tcl commands, such as `string`, `info`, etc. Depending on the nesting level, an arbitrary number of dispatch levels may be achieved. However, nesting objects for creating extensible submethods is always limited to the per-object scope. Hence, classes may not provide such submethods to their instances or have them inherited through their subclasses. Also, nested objects are subjected to the recreation of their parent objects. In such scenarios (e.g., the `info` implementation), method delegation is preferable.

To sum up, objects as methods and submethods offer the following advantages: (1) It is possible to register and de-register not only methods (as explained above), but any kind of subcommands under arbitrary names. (2) The registration and de-registration can happen at runtime. (3) The technique can be used recursively. (4) One can equally use interceptors (filters, mixin classes) at the submethod level.

3.3 Generalized Interfaces for Parametrization

A major achievement is the unification and refactoring of the parametrization infrastructure in XOTcl 2.x. Prior to this, the various concerns of parametrization (e.g., specification of parameters, value parsing, default values, validation of values, handling of required parameters, error messages and introspection) appeared scattered over multiple implementations in the C and XOTcl code base. For implementing Tcl commands in C, it is current practice to implement argument checking (number of arguments, handling of value constraints, options etc.) separately for each and every command. Providing consistent behavior and error messages requires redundant work and maintenance efforts. The capabilities of the parameters for Tcl `procs` are, as well, sufficiently different from the C level. Many C implemented commands (like `set`) allow a single optional argument while this is not language-supported for procs, which provide only optional and variable arguments (via `args`) or arguments with default values to express optionality.

3.3.1 Defining, Evaluating, and Applying Parameters

In general, *parametrization of differences* [2] is a key principle of software reuse to express a family of software components through a shared piece of code with built-in variation points. In object-oriented software reuse, different variants of parametrization co-occur (and interact). In XOTcl [25], for instance, the variety of parametrization mechanisms is considerable. They range from parametric operations (i.e., commands, procedures, and methods) to object and class parametrization. The latter are based on the precedence order of classes and allow to set certain instance variables based on a parameter during object initialization. Furthermore, defining object-class and class-class relationships is handled in XOTcl by parametrization (i.e., the superclass of a class is provided as a parameter).

An important aspect of all forms of parametrization is specifying parameters and their properties. A parameter has an *identifying name* and a set of *parameter properties*. The parameter properties realized are:

- requiredness (optional or required parameters),
- placement constraints (positional vs. non-positional parameters; sometimes called named parameters),
- value constraints,
- conversion properties,
- evaluation properties, and
- parameter defaults

A *parameter definition* refers to the definition of a set of parameters with their properties (such as the parameter definition of a method, i.e., its signature). The parameter definition provides constraints about permissible arguments, when a method is called. We refer to the step of validation of arguments as *parameter evaluation* or *argument parsing*.[2]

If a parameter is defined as *required*, the invocation must contain a value in its argument list. If a parameter is declared positional, the according value for the parameter is found via the matching position. If a parameter is *non-positional*, the value is provided together with the parameter name as a pair. A *value constraint* specifies that a subset of the syntactically permissible values is accepted. *Conversion properties* can be used to specify type conversions of the passed values. *Evaluation properties* can be used to perform certain operations during the evaluation of values or defaults. *Parameter defaults* are used when the argument vector did not provide an actual value for a parameter. Instead, a default value is initialized for the parameter consumer.

Argument evaluation receives as its input a parameter definition and an argument vector. During argument evaluation, certain constraints specified in the parameter definition might not be satisfiable, therefore error messages can be produced. After argument evaluation, a transformed argument vector is available that has to be applied on the argument consumer. This *parameter*

[2]Note that, throughout the paper, we refer to parameter values as *arguments*.

application is different depending on the kind of parameter consumer (i.e., a C function, a Tcl procedure body, or an object). For example, object parameters apply to object variables.

Explicit parameter definitions can not only be used for consistent argument evaluation, but also for producing consistent error messages, for facilitating parameter introspection (e.g., by providing the developer with means to reason about valid arguments), and for creating documentation (e.g., by transforming parameter definitions directly).

3.3.2 Parameter Interfaces in XOTcl 1.x and XOTcl 2.x

In XOTcl 1.x, programming interfaces for defining parameters were used at various places for creating different kinds of parametric operations. Also, each of them was equipped with different capabilities and a distinct syntax. XOTcl 2.x comes with a common parameter infrastructure. All parameter definitions are parsed into a common C structure which is used for the following kinds of parametrization:

- Parameters for *C implemented commands*. C implemented Tcl commands are defined by C functions with a common interface. Usually, the called C function is responsible for processing the input `Tcl_Obj` array (i.e., `objv`), for performing the appropriate conversions, and for generating error messages. These tasks must be achieved for each and every C implemented Tcl command separately. Providing consistent behavior and error messages causes redundancy. Introspection is not possible.

  In the XOTcl 2.x infrastructure, the method definitions of all C implemented XOTcl commands are defined in a signature definition language (in Tcl syntax) that supports the parameter properties mentioned above (i.e., requiredness, positional vs. non-positional, typing information, defaults, etc.). From these signature definitions, stub functions in C are generated automatically. These stubs handle the parameter evaluation. The parameter definitions are available at runtime for, e.g., introspection. More details are provided below.

- Parameters of *C implemented methods* are very similar to command parameters, but they have to provide access to the currently active, method-owning object on which the method has to be dispatched. In XOTcl 1.x, every method definition in C had to incorporate this knowledge. In the XOTcl 2.x infrastructure, the parameter definitions for C implemented methods are defined by the same signature definition language as command parameters (see the previous item). The signature definition language allows to specify what kind of stub should be generated (see for example the generation of stubs for C commands via `xotclCmd` and for C methods via `classMethod` in Listing 5 on page 11).

- Parameters of *Tcl implemented methods* are defined in XOTcl 1.x in the `procs` and `instprocs` definitions as an argument after the method name (same as for Tcl `procs`). XOTcl 1.x supports non-positional and positional parameters with value checking on non-positional parameters. Non-positional parameters can be defined as required and can have defaults. Since non-positional parameters are not supported natively by Tcl, XOTcl converted the argument list to Tcl's vararg interface (using `args`) internally and inserted a call to a specific handler into the procedure body (i.e., `::xotcl::interpretNonpositionalArgs $args`).

In XOTcl 2.x the method parameter definition is provided as an argument block following the method name. The definitions are now fully orthogonal and allow the same set of parameter properties (e.g., value constraints on positional and non-positional parameters, optional positional parameters, etc.). The same infrastructure for argument evaluation as for the other parameter types is used (see Section 3.3.3 for more details).

- *Object parameters* provide means for the parametric configuration of objects during their creation. The applicable parameters are determined by the precedence order of the classes (i.e., the class hierarchy and the mixin classes). Early versions of XOTcl provided the `-parameter` interface for defining the parametric configuration of class instances, starting with 1.5 slots [25] became available. The `-parameter` interface of XOTcl 1.x supported rich parameter properties, however, they were substantially different to properties for method parameters (e.g., deviating default handling and value checking). Besides, the `-parameter` interface did not support required object parameters.

  In the XOTcl 2.x infrastructure, the infrastructure for parameter evaluation is shared by object parameters. Note that the parameter definitions are purely declarative, while the XOTcl 1.x object parametrization was procedural. That is, every word starting with a dash was interpreted as a method name. In XOTcl 2.x, the object parameter definitions are collected, for instance, from the slot definitions into a declarative structure used for argument evaluation. To provide backwards compatibility, XOTcl 2.x adds an `args` at the end of the parameter definition which is evaluated following XOTcl 1.x semantics.

3.3.3 Defining Parameters in XOTcl 2.x

The XOTcl 2.x infrastructure supports a canonical parametrization model and processing infrastructure, based on a set of components and artifacts shared by command, method, and object parameters. These components and artifacts distribute over two binding times in parameter handling, i.e., defining parameters and argument evaluation at invocation time. While the parameter definitions for C implemented command and methods are processed during the compilation of XOTcl, the parameter definition for Tcl implemented methods and object parametrization is situated at the scripting level. In the next few paragraphs, we discuss the parameter handling for C implemented methods. Then, Tcl implemented methods and object parametrization are addressed.

Parameters for C implemented Commands and Methods As Figure 1 shows, the definitions of the signatures for the C implemented commands and methods are provided by parameter definitions stored into a separate file. This file (i.e., `gentclAPI.decls`) is processed by a C code generator into a source code file containing the stubs and definitions (essentially the `XOTclParsedParam` structures) for all defined functions. The generated file is included into the XOTcl C source.

At runtime, the stub is used to invoke the C implemented commands and methods. When such a function is called, the argument evaluator processes the provided arguments against the parameter definitions. During argument evaluation, all basic type conversions, value constraints and default

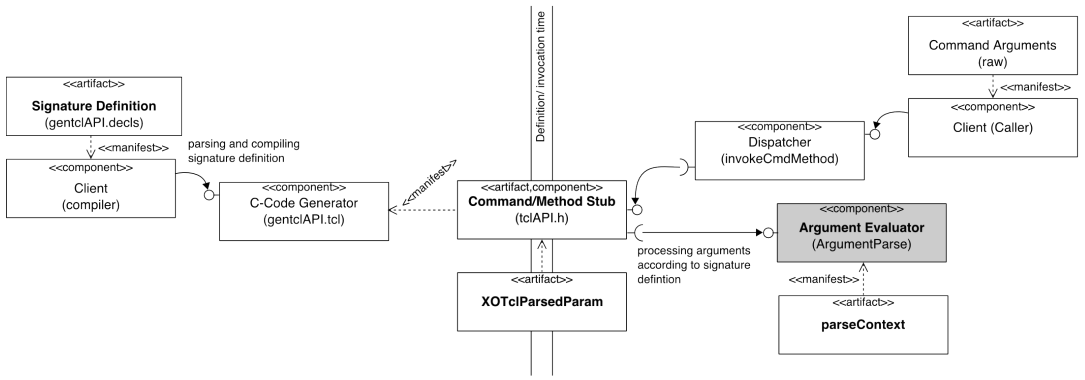

Figure 1: Handling of C Implemented Commands and Methods at Definition and Invocation Time

value handling are performed. The result is called the `parseContext` which, essentially, manages the raw and converted arguments in a canonical manner.

```
#
# XOTcl commands
#
xotclCmd alias XOTclAliasCmd {
  {-argName "object" -required 1 -type object}
  {-argName "methodName" -required 1}
  {-argName "-objscope"}
  {-argName "-per-object"}
  {-argName "-protected"}
  {-argName "cmdName" -required 1 -type tclobj}
}
...
#
# class methods
#
classMethod alloc XOTclCAllocMethod {
  {-argName "name" -required 1}
}
```

Listing 5: Parameter Definitions for C implemented Commands and Methods

Listing 5 provides an example of the parameter definitions for the XOTcl command `::xotcl::alias` and the XOTcl method `alloc`. The parameter definition for the C implemented command is defined as `xotclCmd`, while the definition for the C implemented method is marked as `classMethod` (i.e., a method for the meta-class `Class`). A structure of nested Tcl lists is used for the language's syntax. Besides the parameter name `argName`, several parameter properties such as `required` and `type` settings are provided in this example. The `type` property is not only used for value checking, but as well as a conversion property.

```
static int
XOTclAliasCmd(Tcl_Interp *interp, XOTclObject *object, char *methodName,
                              int withObjscope, int withPer_object, int
                              withProtected, Tcl_Obj *cmdName);
```

```
static int
XOTclCAllocMethod(Tcl_Interp *interp, XOTclClass *cl, char *name);
```

Listing 6: Generated C Function Prototypes for Commands and Methods

The C code generator produces function prototypes for the command and method stubs. Listing 6 gives the prototypes for the signature definitions above. The listing shows that the current interpreter is passed as the first argument. Non-positional parameters are converted per default into separate integer arguments carrying the prefix `with`. XOTcl objects are passed in their internal representation. For C implemented methods the class or object is provided as the second argument.

```
static int
XOTclCAllocMethodStub(ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj
    *CONST objv[]) {
  parseContext pc;
  XOTclClass *cl = XOTclObjectToClass(clientData);
  if (!cl) return XOTclObjErrType(interp, objv[0], "Class");
  if (ArgumentParse(interp, objc, objv, (XOTclObject *) cl, objv[0],
                     method_definitions[XOTclCAllocMethodIdx].paramDefs,
                     method_definitions[XOTclCAllocMethodIdx].nrParameters,
                     &pc) != TCL_OK) {
    return TCL_ERROR;
  } else {
    char *name = (char *)pc.clientData[0];
    parseContextRelease(&pc);
    return XOTclCAllocMethod(interp, cl, name);
  }
}
```

Listing 7: Generated Stubs for Argument Parsing and Error Handling

Finally, the C code generator provides a stub based on every signature definition. In Listing 7, we show the stub for the `alloc` method. We learn that the argument evaluator `ArgumentParse` obtains its definition from the global table `method_definitions` and provides it output as a `parseContext` structure on success. Finally, the implementation of the `alloc` method is called with arguments compatible with the signature shown above.

Tcl implemented Method Parameters Tcl implemented methods are defined solely at the scripting level (at runtime). The provided parameter definitions are transformed into the common canonical structure `XOTclParsedParam` upon definition time of the method. This structure is saved in the C structure representing the XOTcl method (more specifically, in the `deleteData` of the `Tcl_Command` structure). At invocation time, the parameter definitions are efficiently retrieved from this structure and passed on to the argument evaluator, along with the actual `Tcl_Obj` argument vector (i.e., `objv`). The resulting `parseContext` then serves for supplying the standard Tcl argument handling within the current call frame scope (for more details, see Figure 2). The argument application happens in Tcl as usual via `InitArgsAndLocals()`.

Listing 8 contains an example with XOTcl's parameter definitions. The syntax of the parameter definitions is strongly influenced by OpenACS [18]. The example defines an unconstrained parameter `a` followed by an optional, non-positional parameter `trace`, a positional parameter `b` with an

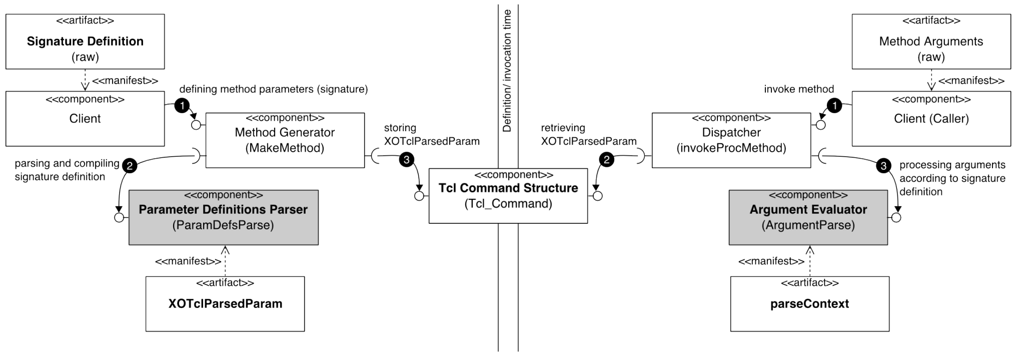

Figure 2: Handling of Method Parameters at Definition and Invocation Time

integer value constraint, and an optional positional parameter `c`. XOTcl users might notice that the requirement of defining non-positional ahead of positional parameters has been lifted. Optional, positional tail parameters as commonly found in Tcl commands (e.g., `set` command, match patterns for `info` commands, etc.)

```
Class create C
C method foo {a {-trace false} b:integer c:optional} {...}
```

Listing 8: Method Signature for a Tcl implemented Method

Object Parameters The parametric configuration of objects shares the same implementation but differs from Tcl implemented methods in its semantics. While for method parameters, the full parameter definitions are provided explicitly, the definitions for object parameters are assembled from configuration values of the classes along the precedence order. Note that the precedence order can be changed dynamically in XOTcl (by defining different per-class mixins or altering the class hierarchies). Therefore, the object parameter definitions might change during runtime of a script.

In XOTcl 2.x, we use a scripted method named `objectparameter` that processes the slots of classes and, so, assembles a parameter definition in the method parameter notation (see previous paragraphs). With this, the same parameter parser is used to compile the computed parameter definitions into the canonical C structure. The same XOTcl 2.x framework can be used for different OO dialects by providing their own implementation of `objectparameter`. Technically, the parameter definition time denotes the moment of calling a `configure` method. The parsed parameter definitions are then cached internally in the class structure for efficient access at subsequent invocation times (i.e., upon creating further offsprings of a class).

```
Class create Foo -parameter {a:integer,required {trace:boolean false}}
...
# Computed parameter definition:
# -a:integer,required {-trace:boolean false} -mixin:relation -filter:relation
#        -class:relation args
...
```

```
Foo create f1 -a 123
```

Listing 9: A Backward-Compatible Parameter Definition for Object Configuration

Listing 9 gives an example of a class `Foo` which provides object parametrization for its future instances. The parameter definition is here specified in the XOTcl 1.x `-parameter` style for backward compatibility. The comment shows the output of `objectparameter`. Note, that the object parameter definition contains as well relation-specific entries for objects, namely `mixin`, `filter`, and `class`.

4 Redesign of Internals

The sections above covered changes in XOTcl 2.x which are conceptually different from XOTcl 1.x and which are visible as extended or novel language features. In the following, we outline selected changes in the implementation of XOTcl 2.x. These internal modifications make use of the improved Tcl 8.5 core infrastructure. We limit ourselves to discussing two selected topics shortly, namely stack management and the object deletion logic.

4.1 Stack Management

An important area of change is *stack management.* XOTcl 1.x provided its own stack management for keeping invocation-specific information on the stack (e.g. name of the actual class or object, method names, stack frame types, etc.). This meant synchronizing two stacks, even under situations where a Tcl stack frame pushed is not paired by an XOTcl frame (e.g., in Tcl procedure invocations) or vice versa (e.g., invocations of certain C implemented methods). Additional complexity was caused by realizing callstack transparency for upvar and the uplevel operations (to provide consistent behavior when interceptors like mixins or filters are added).

In XOTcl 2.x, an integrated callstack management with Tcl 8.5 is adopted. Tcl 8.5 introduces the long desired `clientData` in stack frames which allows to construct new call frame types. XOTcl defines three types of stack frames: *object frames* to make instance variable appear as local ones, *method frames for C implemented methods*, and *method frames for Tcl implemented methods.* The primary advantage of this modification is a unified stack management (e.g., size management) and the avoidance of search efforts to link from XOTcl stack frame contents to Tcl ones, and vice versa. Certain stack operations now turn out slightly more expensive since the search for XOTcl stack frames must account for interleaving Tcl frames. Also, XOTcl exerts now less control about the life-time of a stack frame (e.g., `TclObjInterpProcCore()` performs an implicit pop operation). The original XOTcl-specific stack infrastructure is just kept for Tcl 8.4 backwards compatibility (which might be dropped finally).

4.2 Object Deletion Logic

The design of the deletion logic of XOTcl distinguishes between the explicit and implicit destruction of objects. An *explicit destroy* operation occurs when a script invokes the `destroy` method of an object explicitly (e.g., `$obj destroy`). An *implicit destroy* occurs when, e.g., a Tcl namespace containing XOTcl objects is deleted. Or, when Tcl issues the C- level `Tcl_DeleteCommandFromToken()`. This function is called in various situations, such as creating Tcl commands with the name of an XOTcl object, or renaming Tcl commands to an empty string. For these implicit cases, it is essential that the object is actually deleted, regardless of whether the provided `destroy` methods propagate to the top-level object `destroy`.

XOTcl 2.x performs the deallocation of objects fully symmetrically to their allocation. Objects are always created by the class-level `alloc` method, and they are deallocated by the class-level `dealloc` method. The object-level `destroy` finally delegates to `dealloc`. Implicit destroy operations call `dealloc` which, in turn, calls the user-level `destroy` method, if needed.

The time of physically freeing object structures has changed in this process. In XOTcl 2.x, an explicit activation count is maintained for object and classes. Object or classes, which are active on the stack, are never physically destroyed. Upon pop operations, the activation count is checked. When the activation count reaches zero, the objects flagged for deletion are deallocated. The explicit activation count simplified the deletion logic significantly.

5 Evaluation

In this section, we report on a comparative evaluation of selected tasks in the XOTcl object system, namely object creation, destruction, and method dispatches. We compare the XOTcl 2.0.0 development version with XOTcl 1.6.0. The choice of the latter is due to the availability of profiling results previously published (see [23]). All profiling tests are performed under Mac OS X on a 2.8 GHz Intel Core 2 Duo processor. The programs were compiled with gcc 4.01 with `-g -Os` and linked against the same Tcl 8.5.7 library.

5.1 Creation and Deletion of Objects

The first test evaluates the performance of *object creation and deletion* in four variations: pure object allocation (without constructors), object creation (with constructors), object creation with parametric configuration, and object destruction. For the first test, XOTcl allows the generation of objects without constructors via the method `alloc`. The second and third test assess creating an empty object and creating an object with a single instance variable initialized during creation, respectively (using the class `Foo`; see Listing 10). The TclOO equivalents are straightforward, i.e., `alloc` was emulated via `object create`.

```
set c 100000
set ::i 0; run {::xotcl::Object alloc f[incr ::i]} $c
set ::i 0; run {f[incr ::i] destroy} $c

Class create Foo
set ::i 0; run {Foo create f[incr ::i]} $c
set ::i 0; run {f[incr ::i] destroy} $c

Class create Foo2 -parameter {{x 1}}
set ::i 0; run {Foo2 create f[incr ::i]} $c
set ::i 0; run {f[incr ::i] destroy} $c
```

Listing 10: Object Creation and Deletion

The results are summarized in Table 1. For every test, the execution time in microseconds is given (lower is better). The results show that XOTcl 2.0.0 is significantly faster than XOTcl 1.6.0, especially when variables are allocated and initialized during object creation. XOTcl 2.0.0 is more than twice as fast as XOTcl 1.6.0. The tests also show that XOTcl 2.0.0 outperforms TclOO 0.6 in these profiling results, especially for object deletions (i.e., by a factor of more than 15).

Object System	alloc (μs)	create empty (μs)	create with var (μs)	destroy (μs)
XOTcl 1.6.0	4.23	7.89	13.59	3.57
XOTcl 2.0.0	3.69	5.00	5.90	2.95
TclOO 0.6	7.87	5.36	6.32	50.97

Table 1: Object Creation (without Constructor, with Constructor, with Parametrization) and Destruction

5.2 Dispatch Performance

The second test sets out to evaluate the *dispatch performance* in dependence of argument evaluation. The performance is measured in terms of calls per second. We compare method dispatches with zero arguments (`args0`), with three positional parameters (`args3`), with two non-positional parameters (`np2`), and with a mix of two non-positional and three positional parameters (`np2args3`). While the invocation for `np2` is performed without actual argument (the default values are taken), the invocation for `np2arg3` specifies an argument for every parameter (see Listing 11).

```
Class create C
C method args0 {} {return 1}
C method args3 {x y z} {return $x}
C method np2 {{-a 10} {-b 100}} {return $a}
C method np2args3 {{-a 10} {-b 100} x y z} {return $x}

C create c1
cps {c1 args0}
cps {c1 args3 1 2 3}
cps {c1 np2}
cps {c1 np2args3 -a 20 -b 200 1 2 3}
```

Listing 11: Method Dispatch

Table 2 gives the number of dispatch operations per second (higher is better). Again, the results show that XOTcl 2.0.0 performs significantly better than XOTcl 1.6.0. The empty dispatch without arguments is about twice as fast, the dispatch of `np2arg3` is even more than 4 times faster. Also, XOTcl 2.0.0 provides a higher dispatch throughput than TclOO 0.6. We have not implemented non-positional parameter handling, natively unavailable for TclOO. A Tcl-only implementation would incur a significant overhead.

Object System	args0	args3	np2	np2args3
XOTcl 1.6.0	771,968	693,450	336,866	184,997
XOTcl 2.0.0	1,437,876	1,268,713	1,175,979	882,530
TclOO 0.6	1,264,366	1,119,037	n.a.	n.a.

Table 2: Method Dispatch Throughput per Second

5.3 Memory Consumption

Finally, we compared the *memory consumption* per object (see Table 3). It is not sufficient to compare the size of the object structures in the C program, since this will not account for pointers to other Tcl structures like commands, namespaces and hash tables. Hence, we measured the gross effect of creating objects in a running system (for the hardware settings, see the introduction to this section). The test creates one million objects and compares the process size before and after the creation of these objects. It divides the size by the number of objects. For every system, the computed bytes per object are given (lower is better). The results show that XOTcl 2.0.0 uses about

Object System	memory per object (bytes)
XOTcl 1.6.0	450
XOTcl 2.0.0	514
TclOO 0.6	1473

Table 3: Memory Footprint per Object

10% more memory per object than XOTcl 1.6.0. The differences are most likely due to a more eager conversion of `Tcl_Objs` to Tcl commands in XOTcl 2.0.0.

5.4 Code Size

By generalizing the parametrization and factoring out common code, the size of the C code was reduced by more than 2,500 LOC (about 20%). Yet, the functionality was extended significantly and the performance could be improved (see above). The current version contains support for Tcl 8.4 and Tcl 8.5 with 8.4 compatibility requiring a significant code share (e.g., using client data on the stack, function dispatch, variable handling). Dropping support for 8.4 would certainly entail a further reduction in code size.

6 Summary

The Extended Object Tcl (XOTcl) language has diffused across the various Tcl communities for nearly ten years. In this time frame, it has been successfully adopted as a development platform for a variety of commercial, as well as academic applications and application frameworks. Likewise, it was subjected to research on language and application engineering. We started this paper with an overview of how language concepts such as mixin classes and filters were refined over time and how new features such as method delegation, non-positional parameter passing, and slots entered the XOTcl language. This is explained against the background of XOTcl's adoption in numerous research- and industry-driven software development projects.

In this paper, we expounded the details of the modifications and enhancements to XOTcl in its 2.x branch. First, we provided hands-on examples for language-oriented programming, i.e., creating object systems based on bare objects and foundational relations between them, assembling object behavior from command and method sets, and using dispatch bypassing. Second, we presented the use of language-programming primitives, method delegation, and object aggregation to bind objects as message receivers (methods) to other objects, allowing for new forms of behavioral refinements. Third, we outlined the core ideas of a uniform programming model and combined infrastructure for defining, evaluating, and applying command, method, and object parametrization in XOTcl 2.x. Finally, we discussed two major internal refactorings. On the one hand, XOTcl 2.x comes with a callstack management largely integrated with the Tcl 8.5 callstack infrastructure. On the other hand, the overall deletion behavior for the XOTcl object system and its inhabitants was refined. A profiling-based evaluation showed that the XOTcl 2.x branch appears superior to XOTcl 1.6.0 in terms of execution times for creating and destroying objects (under different initialization loads) and in terms of call throughput for method dispatches (again, under different argument evaluation requirements).

The contributions of this paper are threefold: (1) To begin with, the paper introduces newer metalevel programming features (e.g., object system specification and method bindings) and illustrates that they emerged from the continued refinement of the XOTcl 1.x metaobject protocol. However, there are also noteworthy limitations to the metalevel programming possible. The arsenal of available Tcl commands (e.g., `set`, `append`, etc.) can only be used as methods within certain boundaries. Also, parameter introspection under method delegation (i.e., applying `info` on the delegation source on behalf of the target) is constrained. (2) Also, we exposed our motivation to increase the orthogonality between existing (e.g., higher-level vs. lower-level parametrization interfaces) as well as between existing and new language features (e.g., parametrization and parameter introspection). Hence, the XOTcl 2.x branch benefits from an improved separation of concerns. For instance, object parametrization has been decoupled from higher-level features such as slots and the `-parameter` interface. (3) We could demonstrate that these feature consolidations have led to a more maintainable code base. At the time of writing, parametrization stubs for more than 100 C implemented XOTcl commands and methods are generated, including adapters for backward compatibility. While this number will decrease due to continued consolidation, refactoring and maintenance efforts were so substantially reduced. Also, the improved maintainability pairs with an enhanced performance profile of the XOTcl 2 object system. While a set of conceptual and implementation issues remains to be addressed, the XOTcl 2.x branch is, to this date, in a functional and conceptually sound shape. We anticipate an initial release by the end of 2009. The state of development achieved so far serves as a solid base for further-developing XOTcl 2.x towards a framework for language-oriented programming.

References

[1] Archiware. http://archiware.de/, last accessed: September 8, 2009.

[2] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley Longman Publishing Co., Inc., 6th edition, 2000.

[3] DLTK. http://www.eclipse.org/dltk/, last accessed: September 8, 2009.

[4] Donal K. Fellows. OO My! Object Orientation in Tcl Core. Talk at the 12th Annual Tcl/Tk Conference, Portland, Oregon, USA, October 2005. URL http://www.tcl.tk/community/tcl2005/abstracts/Core/oomy.txt.

[5] Donal K. Fellows, Will Duquette, Steve Landers, Jeff Hobbs, Kevin Kenny, Miguel Sofer, Richard Suchenwirth, and Larry W. Virden. Object orientation for tcl. TIP#257, September 2005. URL http://www.tcl.tk/cgi-bin/tct/tip/257.html.

[6] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, last accessed: July 7, 2009, 2005.

[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, October 1994.

[8] Gregor Kiczales, J. Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[9] G. Neumann and U. Zdun. Filters as a Language Support for Design Patterns in Object-Oriented Scripting Languages. In *Proceedings of COOTS*, San Diego, California, USA, May 1999.

[10] Gustaf Neumann. Adding an extensible object system to the core. TIP#279, October 2006. URL http://www.tcl.tk/cgi-bin/tct/tip/279.html.

[11] Gustaf Neumann and Stefan Sobernig. XOTcl for OpenACS: An Introduction to XOTcl and the basic infrastructure of xotcl-core. Tutorial given at the International OpenACS and DotLRN Conference: International Conference and Workshops on Community Based Environments, Antigua, Guatemala, February 2008. URL http://nm.wu-wien.ac.at/research/publications/b719.pdf.

[12] Gustaf Neumann and Stefan Sobernig. Learning XoWiki: A Tutorial to the XoWiki Toolkit. Tutorial given at the International OpenACS and DotLRN Conference: International Conference and Workshops on Community Based Environments, Antigua, Guatemala, February 2008. URL http://nm.wu-wien.ac.at/research/publications/b717.pdf.

[13] Gustaf Neumann and Stefan Sobernig. XOTcl 2.0 – A Ten-Year Retrospective and Outlook. In *Proceedings of the Sixteenth Annual Tcl/Tk Conference*, Portland, Oregon, October 2009. Tcl Association. URL http://nm.wu-wien.ac.at/research/publications/b806.pdf.

[14] Gustaf Neumann and Mark Strembeck. Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scriptling Language. In *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS)*, pages 58–67, Philadelphia, USA, November 2001. URL http://wi.wu-wien.ac.at/home/mark/publications/ccs01.pdf.

[15] Gustaf Neumann and Uwe Zdun. Enhancing Object-Based System Composition through Per-Object Mixins. In *Proceedings of the 6th Asia-Pacific Software Engineering Conference (APSEC'99)*, pages 522–529, 1999.

[16] Gustaf Neumann and Uwe Zdun. Implementing Object-Specific Design Patterns Using Per-Object Mixins. *Proceedings of Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99)*, August 1999.

[17] Gustaf Neumann and Uwe Zdun. XOTcl – an Object-Oriented Scripting Language. In *Proceedings of the 7th USENIX Tcl/Tk Conference (cl2k)*, Austin, TX, USA, 2000.

[18] Open Architecture Community System (OpenACS). http://openacs.org/, last accessed: April 13, 2008.

[19] Harold Ossher and Peri L. Tarr. Using Subject-Oriented Programming to Overcome Common Problems in Object-Oriented Software Development/Evolution. In *ICSE*, pages 687–688, 1999.

[20] Stefan Sobernig. A Gentle Introduction to XOTcl SOAP. Tutorial given at the 6th International OpenACS and DotLRN Conference: International Conference and Workshops on Community Based Environments, Antigua, Guatemala, February 2008. URL http://nm.wu-wien.ac.at/research/publications/b725.pdf.

[21] Mark Strembeck. A role engineering tool for role-based access control. In *Proceedings of the 3rd Symposium on Requirements Engineering in Information Security*, Paris, France, August 2005. URL http://wi.wu-wien.ac.at/home/mark/publications/sreis05.pdf.

[22] Mark Strembeck. Engineering Dynamic Policy Based Systems – A Policy Language Based Approach. Habilitation thesis, Institute of Information Systems and New Media, Vienna University of Economics and Business Administration, April 2008.

[23] TclOO. http://wiki.tcl.tk/18152, last accessed: October 5, 2009.

[24] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of OOPSLA'87*, pages 168–182, 1987.

[25] XOTcl – Extended Object Tcl. http://www.xotcl.org/, last accessed: May 6, 2008.

[26] Uwe Zdun. Patterns of Argument Passing. In *Proceedings of the 4th Nordic Conference of Pattern Language of Programs (VikingPLoP2005)*, pages 1 – 25, Otaniemi, Finland, 2005. URL http://www.infosys.tuwien.ac.at/Staff/zdun/publications/arguments.pdf.

[27] Uwe Zdun. Pattern-Based Design of a Service-Oriented Middleware for Remote Object Federations. *ACM Transactions on Internet Technology*, 8(3):1–38, 2008.

[28] Uwe Zdun, Mark Strembeck, and Gustaf Neumann. Object-Based and Class-Based Composition of Transitive Mixins. *Information and Software Technology*, 49(8):871–891, 2007.